C. P. U. Bach:
Using Markov Models for Chorale Harmonization

A Thesis presented

by

Christopher A. Thorpe

to

Computer Science and Music

in partial fulfillment of the honors requirements

for the degree of

Bachelor of Arts

Harvard College

Cambridge, Massachusetts

April 24, 1998

To Δ. Λ.

Contents

Acknowledgments

I would like to thank the following individuals and organizations:

David Lewin, for his interest, motivation, encouragement, understanding, sense of humor,[1] and sincere friendship over the past four years.

Stuart Shieber, for engaging conversations, asking me if he could advise this thesis, and for pointing me in the right direction on the computational issues.

John Stewart, for his contagious love of Bach and for the motivation I obtained from his statement that computers would never be able to write good chorale harmonizations.

Margo Seltzer, for introducing me to Rob Pike's "friend" Mark V. Shaney (who must be a descendant of C. P. U. Bach).

Henry Leitner, for friendship, encouragement, good food and good music.

The above five individuals again, for reading and grading this thesis.

Janis Siggard, for the love of music that I gained from years of piano lessons. Jackie Sorensen, Carla Seely, Lori Labrum, Anna Thompson, Robyn Healey, Margaret Fielding, Randy Orwin, Tim Farr, Hans Baantjer, Jay Andrus, Rockland Osgood, and Heather Knutson for putting up with me through my musical training.

Harry Lewis, for rightly noting that joint concentrations at Harvard are for those interested in the intersection, not the union, of the two fields.

Eleanor Selfridge-Field and the Center for Computer Assisted Research in the Humanities at Stanford University, for making the data available to me.

David Cohen, Ed Gollin, Alan Gosman, and Christoph Neidhöfer for listening to and evaluating the output.

Viaweb, for giving me a job and a quiet place to work on this thesis.

Rocco Servedio, for the "citroids," a listening ear, interesting advice on the problem, and a comfortable sofa.

Adam Deaton, for the Goldberg Variations and the same sofa.

Peg Schafer, for ice cream at late-night hacking sessions.

Tom Kelly, for his interest in and discussion of computer modeling of music.

Robert Levin and Robert Kendrick, for their interest and encouragement.

All my friends, who haven't gotten to see me much for the past few weeks.

Allen, Elizabeth, and Matthew Thorpe, for their continued love and support, and for giving me a childhood filled with music.

[1] Hier wird keine Schweinerei ausgeübt!

Chapter 1

General Introduction

Markov models have been applied with varying degrees of success to many problems, including natural language processing, speech recognition, protein recognition and melody composition. The most common problem with Markov models is that they are highly localized, and do not capture large-scale structure. This thesis applies a Markov model to the problem of harmonizing a melody in the style of J. S. Bach. Because the melody is separate from the harmonization process, it provides a "thread" that exists outside the local realm of the probabilities.

Thus, harmonizing an existing melody is a "completive" rather than a "generative" task: the Markov model simply brings out one of many underlying harmonic structures that are compatible with the melody. The results of a simple-minded Markov model still suffer from the effects of its localized nature. To further improve the large-scale structure of the harmonization, we then use a new Markov model based on rhythmic and metric features of the given melody: we initially create "signposts" for the harmony based on the

rhythmically strongest events, and fill in the rhythmically weaker events using appropriate probabilities.

Other Markov models can be applied with some (though lesser) degree of success to completing the four-part harmony, given a bass line and melody. This work is, however, incomplete and secondary to the primary goal of harmonization: the inner voices are implied to a considerable extent by the melody and bass line. I have included examples and a brief discussion of this work in an appendix for interested readers.

Chapter 2

Introduction for Computer Scientists

This thesis presents an algorithm that creates a bass line to accompany chorale melodies in the style of J. S. Bach. The algorithm is based on a Markov model. While Markov models have been used in the past for other computational problems including speech recognition, protein recognition, and even melody composition, a Markov model has to my knowledge not been applied to the process of harmonization.

Humans have been composing polyphonic music for hundreds of years. In this context, I use "polyphonic" to refer to music in which more than one melodic line is performed simultaneously. Most music heard today outside whistling or singing alone is polyphonic in some sense. The most unique feature of polyphonic music is harmony: how notes performed simultaneously relate to one another.

A complete discussion of harmony is obviously beyond the scope of this thesis. The program presented here attempts to add harmony to a given melody

by imitating the compositions of Bach. Since Bach's chorale harmonizations are frequently considered to be a standard of four-part chorale composition and of Western tonal harmony, his compositions are an appropriate choice for the data set. In analogy to natural language, we might say that Bach is a "fluent native speaker" of chorale harmonizations, and that his music comprises an exemplary corpus of data in this "language."

The use of a formal computational method in application to composition of music presents several unique challenges. Obtaining the data is difficult, as most music is not in a computer-readable form. Furthermore, the encoding with which music is input into the computer frequently requires substantial transformation before being in a format useful for a particular problem. I wrote a program to parse the KERN music data format, in which I obtained the chorales, and another to output the data in a notation compatible with the LISP programs I wrote to harmonize the melody.

Another problem lies in the evaluation of the output. With many computational tasks, it is often possible both to generate and to evaluate the results using the computer. Not so with music, where unless the rules that govern movement from note to note in multiple voices ("voice-leading") are broken, there is no clear "right" or "wrong." Yet some compositions still "sound better" than others. While this may be due in part to reasons such as rhythmic or melodic variation, resolution of melodic tension, or adherence to tradition, past attempts to compose music using rule-based systems have failed because

of inadequately specifying the rules necessary to generate "pleasing" output.

The algorithm is based on statistics from note to note. Since we imitate Bach's note to note relationships we never violate a voice leading rule. So evaluating the results must be based on more subtle criteria. I evaluate the output in the way a native speaker of a natural language might evaluate another person or computer attempting to speak that language. Occasionally there is a clear explanation for why something is wrong; other times it is necessary to simply say, "there is nothing exactly 'wrong' with this; I just don't say that in my language."

Finally, some new ideas developed here as unique to music may be useful in other applications, for example, the top-down Markov model outlined in Chapter 6. While this was particularly effective and easy to implement due to the inherent periodicity and hierarchy of chorale phrases, such a method may prove useful in the future in statistical modeling of other problems that exhibit structure on multiple levels, e.g. natural language or protein folding.

Chapter 3

Introduction for Musicians

This thesis presents an algorithm that creates a model for the harmonization of chorale melodies and uses it to generate bass lines for a given melody. When writing computer programs to compose or analyze music, it is necessary to formalize music in a very mathematical way so that computers can process the information.

In this thesis we represent musical information as pitch classes, durations and metric positions — all very mathematical and quantifiable pieces of information. By examining this information present in the chorale harmonizations of J. S. Bach, it is possible to generate high-quality bass lines using a statistical technique known as a Markov model.

The underlying principle of a Markov model is a set of states and probabilities on transitions between these states. A state might be an English word, a point on a hypothetical journey, or a pair of voices in a chorale. Then, for each state there is a specific probability for the transition to every other state — zero if there is no transition. For example, in a model of a limited subset of English, we might assign the words SEA, LION, and TOPSY the following transition probabilities:

SEA	→	BASS	25%	LION	→	TAMER	15%
	→	LION	15%		→	MANE	25%
	→	BREEZE	50%		→	KING	40%
	→	NOTHING	10%		→	RUNNING	20%
TOPSY	→	TURVY	100%				

This model is called a "first-order" model because a transition to a state depends only on one previous word. A model where the transitions depended on the two previous words is called a "second-order" model; three words a "third-order;" and so forth. The methods outlined in the following chapters for harmonizing melodies use first and second order models.

One crucial observation about Markov chains is that because the transitions are based only on the immediately previous state or states, there is little possibility for large scale structure or even continuity between states outside the "window" of context. For example, the model above allows for the phrase "SEA LION MANE" which isn't likely to occur at all in English, since sea lions don't have manes. Markov chains do capture small-scale idioms quite nicely, however: in English, TOPSY only appears in the context "TOPSY TURVY," reflected in that the only transition from TOPSY is to TURVY.

These small-scale idioms are most prominent in this style of music at cadences.[1] Cadence points require special treatment, and there is a limited set of harmonizations for each two- or three-note melodic fragment at the cadence. For example, for a cadence on `MI→FA`, only the bass lines `(SO LA)→RE` and

[1] See p. 13 for the definition of "cadence" used in this thesis.

DO→FA are allowed by our data set, although there are many more bass lines in our data set that accompany MI→FA away from a cadence point. The fast harmonic rhythm and consistent setting of cadences makes a Markov model excellent for harmonizing them. We use this strength as a starting point for all harmonizations, then overcome its weakness (a lack of large-scale structure and continuity) through a top-down approach.

Chapter 4

The Task

The program takes a melody in a Western major key (Ionian mode) and returns an unfigured bass line that harmonizes it, following the traditions of four-part chorale writing exemplified by J. S. Bach. I chose this task because creating the bass line is generally the first step musicians take in generating a complete four-part harmonization of a melody, and because the bass line implies to a large extent the rest of the harmonization.

I also considered solving the problem by generating a figured bass line or generating Roman numerals to represent the functional harmony. Both of these methods would have required preprocessing the data using rule-based methods to determine what the harmony actually is. In the interest of saving time and giving the program the most "pure" data — notes with no other information — I elected to use only note information in the statistics used to generate the results. A bass line with only note information shapes the harmony strongly without being constrained by rules of figured bass or fundamental bass.

One particular problem in using a statistical approach for harmonization is that Bach wrote a limited number of chorales. So our task is unlike natural language processing, where the amount of usable data is limited more by computational resources than by the amount of data. Of the limited number of chorale harmonizations written by Bach, fewer than half have even been entered into the computer music format I used. Of the approximately 170 chorales available in the database, only 83 are in the Ionian mode. In these chorales, there are 4,080 soprano events in total.

The first problem in using these data is that the chorales were written in various keys,[1] but harmonic function is for our purposes irrespective of the key of the chorale. Simply transposing all chorales into the same key does not provide an adequate solution: due to the limitations of human voices, Bach set different melodies in different keys to make vocal performance possible. Transposing all of the chorales into a common key would result in impossible vocal ranges for some of the voices.

To solve this problem, I chose to define each note not as an actual pitch (as might be played on the piano) but as an exact scale degree in relation to the key of the chorale. I used the standard chromatic solfégè to represent the pitches of the notes, where `DO` represents the tonic note. For example, if the chorale were in the key of D major, `DO` would represent any pitch D in any octave. This solves the problem of transposition and range for the

[1] "key" refers to the most important pitch of the scale in the work.

following reasons: First, because we have discarded octave information, a note is never "out of range" for a particular part. Since the functional harmony is determined by the relationship between the bass and the soprano without regard to octave, discarding octave information does not significantly make the results less meaningful. One caveat is that a musician must infer the octave information from the results, but since the soprano and bass lines are so far apart it is a matter only of ensuring that the bass line remains within a reasonable range.

Before discussing the other problems and their solutions, it is necessary to define several terms which will be used throughout this thesis. In connection with these definitions, the reader should inspect Musical Examples 1-30 on page 14. The examples are arranged in rough order of musical complexity, but will not be referenced in that order, so that we may compare various "events" at different locations on the page. Let us now proceed to the definitions:

pitch The one of twenty-one chromatic solfégè symbols for a note.

For instance:

- `DO`: diatonic first degree of the key
- `RE`: diatonic second degree
- `RI`: chromatically raised second degree
- `RO`: chromatically lowered second degree

duration The amount of time a note is held. For instance:

- `|4|`: quarter note (one quarter of a 4/4 measure)
- `|2.|`: dotted half note (three quarters of a 4/4 measure) [2]

position The rhythmic location a note occupies within a measure, measured in thirty-second notes from the beginning of the measure. The four beats in a 4/4 measure occur on positions 0, 8, 16, and 24.

note A triple containing a duration, a pitch, and a position, or a "continuation" note indicating that the previous note is still sounding. For instance:

- `(|4| SO 8)`: a quarter note on the fifth degree of the key occurring on the second beat of the measure. This event is shown in Music Example 12.
- `(|.| X 16)`: a continuation note

event An ordered set of one or more notes in any one voice. For instance:

- `((|8| FA 0) (|16| SO 4) (|16| LA 6))` (exemplified by Music Example 15)
- `((|4| LA 16))` (Music Example 13)

[2] a single dot extends the duration of a note by half its value.

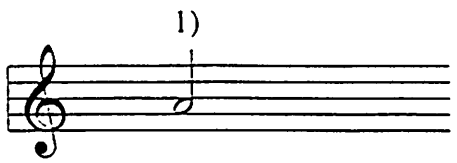
chord An ordered set of two or four events, each of which occurs simultaneously in a unique voice. The position of an event in this set indicates its voice. See Appendix C for examples of chords.

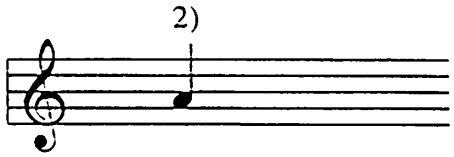
OVC (Outer Voice Chord) A chord consisting of two events. The first consists of exactly one note, not a continuation, and contains the melody (soprano). The second consists of one or more notes and contains the bass line that accompanies the melody. Since occasionally there are more notes in the melody than the bass line, the second event may contain a continuation note to indicate that the soprano moved before the previous bass note had finished.

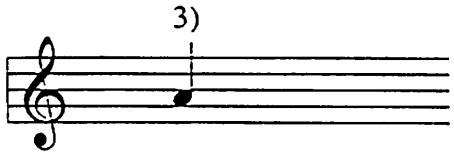
4VC (Four Voice Chord) A chord consisting of four events. The first consists of exactly one note, not a continuation, and contains the melody (soprano). The second, third and fourth events consist of one or more notes and contain the alto, tenor and bass lines that accompany the melody.

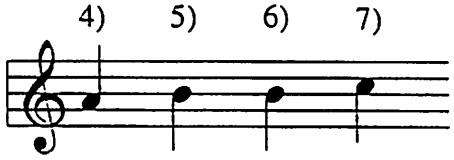
cadence Generally, "cadence" refers to the close of a musical phrase. In this thesis, we use a more specific definition with regard to chorale harmonizations. The conclusion of each line of text set in a chorale is accompanied by some idiomatic harmonic progression approaching a chord that seems more "stable" to the listener. Bach used very similar cadences in his chorales to harmonize similar melodic fragments at the ends of various phrases. A cadence ordinarily comprises the last two or three events of a phrase.

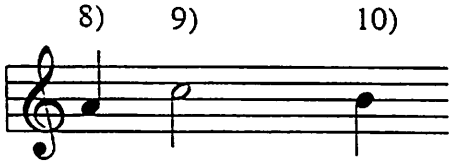
Musical Examples 1-30

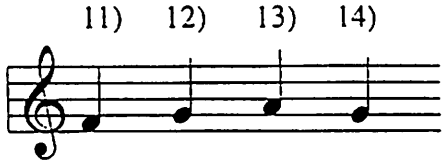


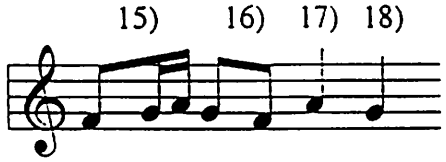


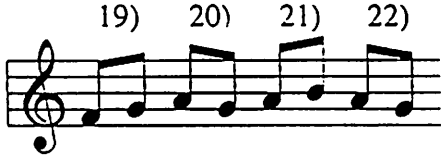


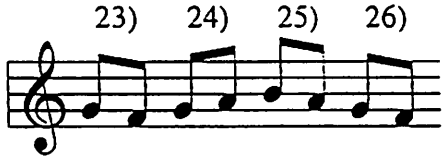


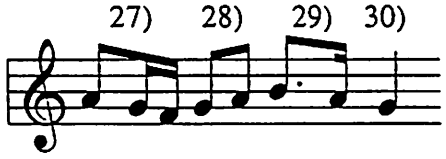












According to the definitions for the notes in the data set, there were 1,181 unique OVC's, and 2,748 unique 4VC's. The problem of data sparsity is serious not because of the small number of chords but because there are so many unique events. With too many unique chords, there are few or no chords corresponding to those in any particular context.

To solve the problem of data sparsity, I considered and discarded traditional "data smoothing" methods. All of these methods require inferring probabilities for data points that do not exist in the original data set. To do this with music would require a set of rules for proper voice-leading between two chords in order to avoid corrupting the data set with mistakes such as parallel fifths or octaves. This is complicated by the fact that parallel fifths or octaves are likely to appear in two consecutive chords unless deliberately avoided, because they are consonant intervals.

Instead, I chose a new approach that looked at the data in a less exact manner. Rather than consider events equal if they were note-for-note equivalent, I implemented various rules for testing equality of events based on their "essence." For example, a bass line may or may not have passing tones between two chord tones on the beat, but the functional harmony of two strong bass notes with a short, weak passing tone in between is equivalent to the functional harmony of the same two strong bass notes with no passing tone. By defining less rigid rules, e.g. rules that allow such events to be "equal" even though they differ slightly, we reduce the data sparsity by increasing the

number of possible events corresponding to those in a particular context.

In order to discuss the equivalency of data points (chords) we must first discuss the equivalency of notes and events.

I defined four rules for defining the equivalency of notes:

Name	Description	Musical Exx. (p. 14)
PITCH-EQ	Only the pitches are equal.	$1 = 2$; $2 = 3$; $7 = 9$ $7 \neq 10$
NOTE-EQ	Both the pitches and the durations are equal.	$4 = 8$; $5 = 10$; $6 = 10$ $1 \neq 2$; $7 \neq 9$
BEAT-EQ	Pitches, durations, and "strong or weak" metric position are equal.	$4 = 8$; $5 = 10$ $6 \neq 10$; $7 \neq 9$
POSITION-EQ	Pitches, durations, and exact position within the measure are equal.	$4 = 8$ $5 \neq 10$; $6 \neq 10$; $7 \neq 9$

After establishing criteria for note-equivalence, we can then develop criteria for event-equivalence. Since events involve one or more notes, our rules for event-equivalence can be used in conjunction with rules for the notes they comprise. Significant and problematic in this context are non-chordal and submetric notes, as two events with the same chord tone in the bass should be judged equivalent even if all of the notes within the event are not equivalent. I defined the following six rules for determining equivalency of events, where TYPE refers to PITCH, NOTE, BEAT, or POSITION (from the preceding table).

Name	Description
TYPE-ALL-EQ	Each note in the first event is TYPE-EQ to the corresponding note in the second event.
TYPE-START-EQ	The first n notes of both events are TYPE-EQ; n is the number of notes in the shorter event.
TYPE-FIRST-EQ	The first note of the first event is TYPE-EQ to the first note of the second event.
TYPE-SET-EQ	Each note of the event with the fewest notes is TYPE-EQ to some note in the other event.
TYPE-ANY-EQ	At least one note of one event is TYPE-EQ to a note in the other event.
TYPE-SIMUL-EQ	Every note in the shorter event is TYPE-EQ to a note occurring simultaneously in the other.

Some examples of such rules:

Rule	Positive Musical Examples From Example Page (p. 14)	Negative Examples
PITCH-ALL-EQ	$25 = 29$; $13 = 17$	$15 \neq 19$; $19 \neq 23$
POSITION-SIMUL-EQ	$24 = 28$; $13 = 17$	$16 \neq 26$; $15 \neq 19$
PITCH-SIMUL-EQ	$15 = 19$; $13 = 21$	$15 \neq 19$; $16 \neq 24$
PITCH-FIRST-EQ	$11 = 15$; $15 = 19$; $16 = 24$	$12 \neq 19$; $12 \neq 23$
NOTE-FIRST-EQ	$12 = 23$; $15 = 19$	$12 \neq 19$
BEAT-FIRST-EQ	$11 = 15$; $15 = 19$; $16 = 26$	$12 \neq 19$; $12 \neq 23$
PITCH-SET-EQ	$11 = 15$; $21 = 29$; $15 = 26$	$16 \neq 24$; $13 \neq 19$

Since there are four tests for note equivalency and six for event equivalency, there are twenty-four possible compositions of these tests. Intuitively, we may omit many tests because they are equivalent to others. For example, POSITION-SET-EQ is equivalent to POSITION-ALL-EQ because POSITION-EQ requires the duration and position of the notes to be equal.

Now that we have rules for determining the equivalency of notes and events, we can determine the equivalency of chords using these rules. I de-

fined several rules for the equivalency of OVC's, of forms TYPE-EQ-SandB and STRONG-TYPE-EQ-SandB. The former tests the soprano events with NOTE-FIRST-EQ (recall that all soprano events have exactly one note) and the bass events with PITCH-TYPE-EQ. The latter tests the soprano events with BEAT-FIRST-EQ and the bass events with PITCH-TYPE-EQ. Since the soprano events and bass events occur simultaneously within any chord, by testing the soprano events with BEAT-FIRST-EQ we verify that both chords are of the same metric strength. Using various chord equivalency functions yielded results of various quality. These results will be described in the following chapters.

Chapter 5

The Simple Approach

I constructed the sets of bi-OVC's and tri-OVC's from Bach's chorale harmonizations by taking all sets of two or three consecutive OVC's. Given these bi- and tri-OVC's, I generated probabilities for the occurrence of a particular OVC following one or two other OVC's, as well as the probabilities for the occurrence of a particular bass note harmonizing a given soprano following one or two other OVC's.

The simple algorithm for harmonizing a phrase begins by examining the last few events in the melody, then assigning a cadence to these events by randomly choosing a cadence from all those Bach used to harmonize these events. This cadence creates a complete harmonization for the last few events in the melody. In Figures 1 and 2 (p. 21), significant melodic events are labeled with Greek letters, proceeding backwards. The cadence events — those harmonized in this stage — are at α, β, and γ.

Following the choice of a cadence, the program works backwards. Each successive melodic event is assigned a bass event using the method shown in Figure 2 and described below:

1. In Figure 2, we see that the melodic events γ and β are now harmonized. We then create a list of all bi- or tri-OVC's in the data that end with the OVC's marked by γ and β. Figure 2, index 1 represents this with the horizontal bracket above the bass notes of γ and β.

2. Extract from this list all OVC's whose first melodic event matches δ. In Figure 2, index 1, this is represented by the slanted line pointing up and left toward the last soprano note (δ) unharmonized in Figure 1.

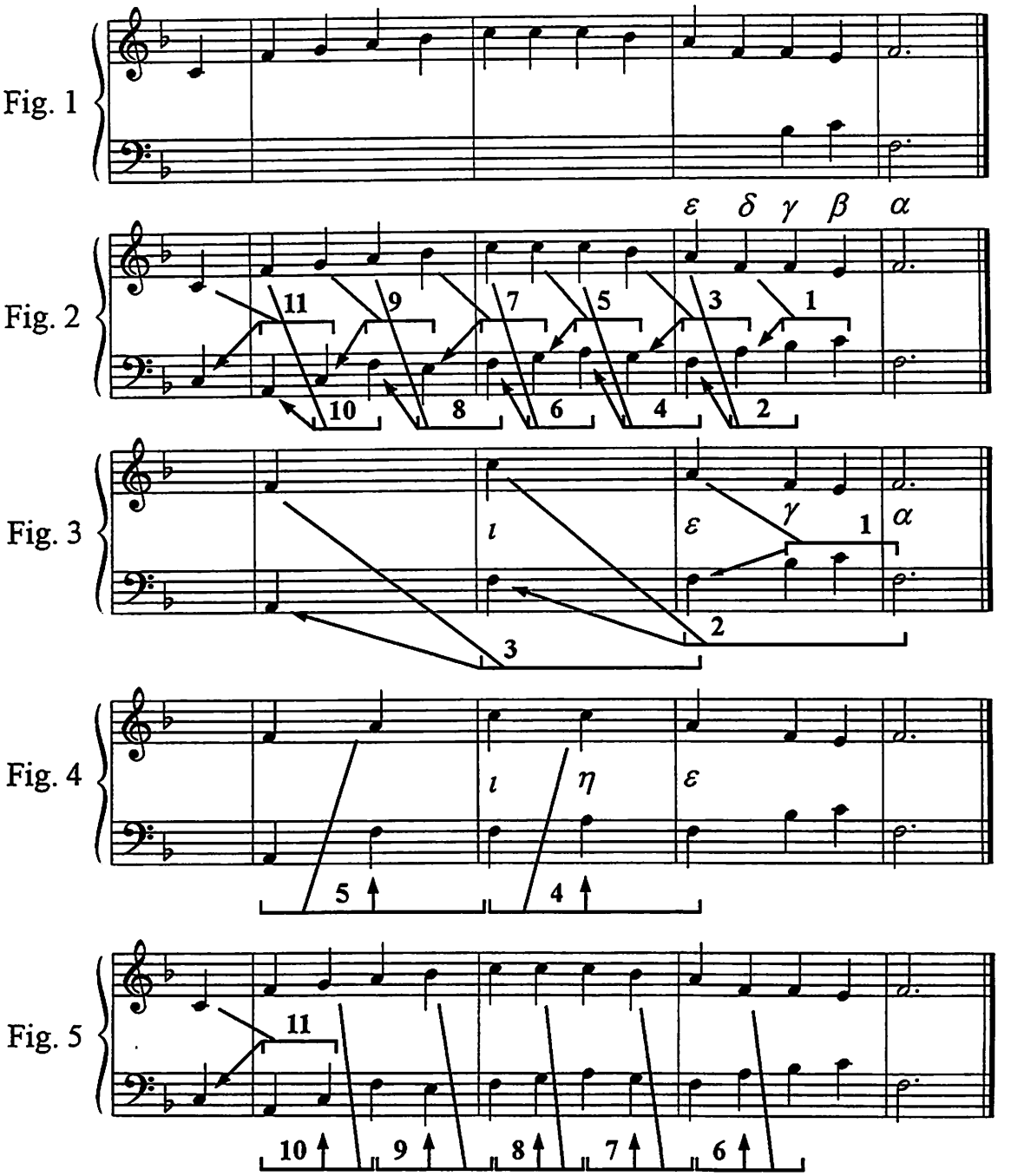
3. The list generated in (2) now contains all OVC's in Bach's works with soprano sequence $\delta \rightarrow \gamma \rightarrow \beta$ and bass sequence as on Figure 2 beneath events γ and β. Choose at random an element from this list. Since duplicates are allowed in this list, we choose with probability equal to the frequency with which Bach used a particular event to harmonize the given soprano sequence.

4. Assign the bass note from the event chosen in (3) to "harmonize" the soprano note δ. The arrow from Figure 2's index 1 pointing to the bass note A represents this assignment.

Repeat steps 1 through 4 above working backwards, one event at a time. Since each new application of steps 1 through 4 results in a new bass note, the process "shifts" one note earlier in the music, using the just-completed event as the first finished event in step 1 above. For instance, Figure 2, index 2 shows how the now harmonized events δ and γ, along with melodic event ε, lead to the bass harmonization of ε. The repetition of this process is manifested in Figure 2 in the successive indices 2, 3, ..., 11, which move leftwards (musically backwards) through the events. The phrase is harmonized in this way, from the cadence to the beginning of the phrase.

Figures 1-5

(Figures 3-5 will be discussed in Chapter 6.)



For many examples, this method yields excellent results. These include LEUCHTET-I.1.a-b (p. 44), LEUCHTET-I.3.b-c (p. 45), and BLEIB-I.4.g (p. 36). For others, however, this approach does not work well, resulting in output that sounds disjointed or repetitious.

BLEIB-I.3.a and b (p. 36) have two clear segments; the first five events are acceptable but the leap[1] to the dominant[2] at the beginning of the second full measure is too strong next to the arpeggiated tonic chord. LEUCHTET-I.2.c (p. 44) sounds disjointed due to its three octave leaps, all of which are approached or left by a leap.

Repetitious events include all seven examples for BLEIB-I.1 (p. 35), which have DO on the first and third beats of the first full measure, and all but one of which begin on DO. Example BLEIB-I.1.g is particularly poor: five of eight events are DO. The repetition of the LA-TI-DO figure in BLEIB-I.4.f is unsatisfactory, particularly since it so strongly emphasizes DO which occupies five of the nine events in this phrase.

There are several reasons that this approach fails. One reason is that the naïve use of the Markov method takes only two elements into consideration: pitch and duration. But the metric position of a note in a measure is also meaningful in the composition of counterpoint. For example, certain intervals and large leaps are more likely to occur in certain metric contexts than in

[1] Movement by "leap" happens when two notes are more than two half-steps apart. Compare to movement by "step," which happens when two notes are adjacent.

[2] The dominant is the fifth degree of the scale, SO. Its use in the bass line strongly propels the harmonic motion toward the tonic, or "home" key of the scale.

others. This is why example BLEIB-I.4.d (p. 36) is unsatisfactory: the second beat of the first full measure is in a major thirteenth (sixth) with the soprano on the tonic, and is approached and left by leap. The sixth is a stable, strong interval, and the tonic-mediant[3] pair is the most stable third in a scale. Leaps also contribute to the emphasis of a note. Weak beats generally need weaker consonances or dissonances to propel the harmonic motion to the next strong beat. Because of the strong consonance of and leaps surrounding this chord, it disrupts the rhythmic balance of the phrase. Examining the data that led to this harmonization proves that it is in an incorrect metric position; the result contained the chord `(((|4| DO 16)) NIL NIL ((|4| MI 16)))`. The 16 (picked up by the soprano from the bass) indicates that the original context of this interval was on the third beat of a measure, a strong beat.

Requiring the program to harmonize the data with attention to the metric position of the notes resulted in more acceptable output, if still not completely satisfactory. The examples in BLEIB-II (p. 37) have no such bothersome chords, but many still exhibit the problem of repetition noted above. For example, the examples BLEIB-II.1.d-f each have at least four events on `DO` and move little throughout the phrase.

This problem is due to the nature of the Markov model: the bass line is created without any large-scale plan. It often wanders about aimlessly, or stays on (or near) the same note for the entire phrase, leading to an uninteresting

[3] the third degree of the scale

harmonization. Our Markov chain gives no context on a scale larger than the two- or three-note "window" with which the harmonization is completed.

Another related issue is the limitation that this method works by "looking" in only one direction (backwards). Even with a second-order approach there is no sense of where the music is "coming from," only where it is "going." To achieve two-way context, it might be possible to begin the phrase in the same way as the cadence is built, and try to work toward the middle from both ends of the phrase. Such a method would not adequately address the compositional problem, however, as the first half would have only past context and the second half would only have future context. The only location in the phrase that would have both past and future context would be in the middle.

Due to these serious problems, it is impossible to achieve consistently satisfactory results with this simple approach. The next chapter describes a more complex model that was able to overcome the limitations of the simple Markov model outlined above.

Chapter 6

The Top-Down Approach

I arrived at a novel solution to the lack of large-scale structure in the composition of the bass line. Since Bach's chorale harmonizations are, for the most part, metrically regular, as are the melodies he harmonizes, determining the most important metric information is straightforward. Using this information, it is simple to look at the chorale at a "higher" level by considering chains of consecutive events at the same or higher metric level (e.g. the first beats of each measure) as Markov chains themselves. I shall call the first beat of each measure its "strongest" beat, and the first and third the "strong" beats. Other events occur on "weak" beats. I then constructed the "strongest-set" of bi- and tri-OVC's by creating chains of the OVC's on the strongest beats of each measure in Bach's harmonizations — the same way that I constructed the original bi- and tri-OVC's. I also constructed the analogous "strong-set" of bi- and tri-OVC's from strong beat to strong beat.

I call the sequence of first beats of each measure the "top level" (Figure 3, p. 26), the sequence of strong beats the "mid-level" (Figure 4), and the sequence of all events the "lowest level" (Figure 5).

Figures 1-5

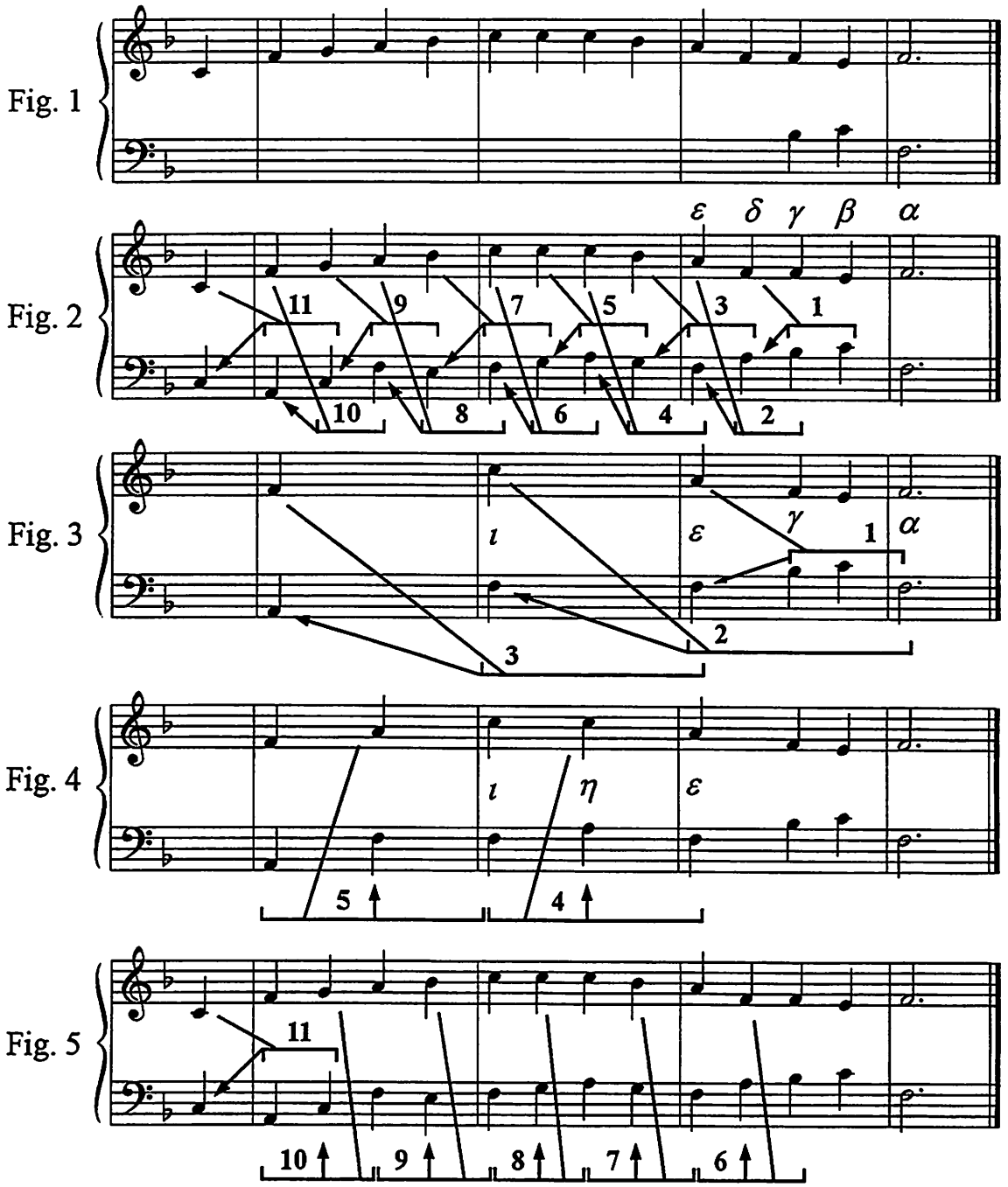


Fig. 1
Fig. 2
ε δ γ β α
11 9 7 5 3 1
10 8 6 4 2
Fig. 3
ι ε γ α
1
2
3
Fig. 4
ι η ε
5 4
Fig. 5
11
10 9 8 7 6

To harmonize a melody, a cadence is first assigned to the end of the phrase (Figure 1). To complete the harmony for the top level, it is first necessary to make sure that the final strongest event has been harmonized in order to have a starting point for the chain. Frequently, it will have been harmonized as part of the cadence. If not, it is given a harmonization using the strong-to-strong probabilities that correspond with the movement from the last soprano event to the strong beat within the cadence. Since all melodies are in either 4/4 or 3/4 time, and cadences are always at least two events long, all final strongest events will either be part of a cadence or lie within one strong beat of a note within the cadence. This assures that there is a valid starting point for harmonizing the strongest notes of the melody.

Each remaining strongest note of the melody is harmonized in exactly the same method used to harmonize the entire phrase in "The Simple Approach" outlined in Chapter 5. A method using the last two strong (not strongest) beats is often necessary to harmonize the penultimate strongest event since there is only one strongest event following. Figure 3, index 1 shows how the events marked α and γ, harmonized as part of the cadence, generate the harmony for the melodic event ε. The structure at index 2 shows the generation of harmony for the melodic event at ι from the OVC's at α and ε.

This process results in a "skeleton" for the harmonization: there is a clear harmonic movement from measure to measure copied directly from Bach's own measure-to-measure harmonic movement. Since most phrases are no longer than three or four measures, there is not enough room for the Markov chain to get "lost" — to reinterpret the current position as one of another unrelated

context — in the way that was possible when the phrases were sixteen events long.

After this skeleton is created, the algorithm steps one step closer and looks at all the strong beats. This results in a sequence of events where every other note is already harmonized. The program can then fill in the harmony between the already harmonized notes. Figure 4, index 4 shows how harmonized events ε and ι, on either side of the melodic event η, generate its bass event.

There are two significant advantages to this method. First, the context is now two-way: each note is filled in based on the immediately previous and immediately following notes. Second, the context is now completely local: there is no "chain" possible because each unharmonized note appears surrounded by already harmonized notes. This completely prevents the problem of geting "lost" in the chain.

To fill in the note, if a tri-OVC exists for which the first and third events correspond to the events before and after the target note, it is assigned the harmonization based on the second event of that tri-OVC. If no such event is found, then the program attempts to find two tri-OVC's that fit the data of forms 1 and 2 following:

1. A tri-OVC whose first and second events correspond to the two events preceding the target and whose third event's soprano corresponds to the target's.

2. A tri-OVC whose second and third events correspond to the two events following the target and whose first event's soprano corresponds to the target's.

If no two such OVC's are found, then an analogous attempt is made using bi-OVC's. If the note is the first note in the phrase (and has context on only one side) then the method to harmonize it uses the same one-way context that was used in the simple approach.

This creates a skeleton of strong beats. After that, it is possible to harmonize the rest of the chorale in the same way, filling in each note based on the tri-OVC's and/or bi-OVC's from the complete data set (Figure 5). The context is again local and bidirectional.

The results from this approach are significantly better than those of the simple method. However, there are still problems. Specifically, there are events that do not make sense in their context. The reason for this is the same as the reason for one problem with the simple approach: certain melodic and harmonic events tend to happen in certain metric positions. Since the top-down approach, while based on metric positions, did not take into consideration whether the previous and next events were stronger or weaker than the target event, it allowed events to creep in that would not otherwise have been permitted. An example of this is the fifth between the soprano and bass on the second beat of the first full measure in LEUCHTET-III.3.c and e (p. 48). It is approached by leap, emphasizing this interval on a weak beat. The original data in LEUCHTET-III.3.c correctly had a weak bass note on beat 2 but also had a weak bass note on beat 1, causing the mistake.

The solution to this problem is to use metric position when comparing bi- and tri-OVC's to the context of the target note. If their metric pattern (e.g. STRONG $\rightarrow$ WEAK or WEAK $\rightarrow$ STRONG $\rightarrow$ WEAK) corresponds to the

context of the target note, then they are acceptable (assuming the pitches and durations also correspond). By doing this, the program does not place an event in the bass line that is uncharacteristic for its metric position, e.g. a dissonance or large leap.

As the constraints on allowed harmonizations become more severe, the program becomes more prone to failure. Since I defined several rules for event equality, I tested several of these rules for bass equivalence on the example melodies. Some rules worked on all chorales, but did not yield results with as high a quality as other rules that did not work on all chorales. For example, PITCH-SET-EQ worked for many chorales but not all, including BLEIB phrase 2. I wanted to use it because it superior results for the other phrases in BLEIB.

My solution to this was to redefine the function that generated each new chord given context to accept multiple test functions. If it was unable to find any chords in the given context using the primary equivalency function, it would then test using the remaining equivalency functions until one was found that yielded a result or the functions were exhausted. This allowed me to order them to achieve the best results.

Choosing where to use the various functions was also an interesting problem. For example, it would have been possible to attempt to harmonize the entire chorale using a certain equivalency function before moving on to another. However, allowing each chord to be generated with the best equivalency function gave the program greater flexibility; it was able to find a greater number of harmonizations by applying different rules to complete a particular context.

If one rule didn't work in one part of the phrase, it could still work well in another part.

The final problem stems from the solution to the data sparsity problem. Since I used a "fuzzy" equality for bass and soprano notes and harmonized in this top-down approach, occasonally a bass event made sense in the higher-level context in which it appeared in the data set but did not work with its surrounding low-level context. When the intervening note was filled in between this bass event and the next strong event, the bass line no longer made melodic sense because the event in the original input went somewhere else. This problem is illustrated by LEUCHTET-IV.1.e (p. 50), in the first two beats of the first full measure. The top-level structure created the `(TI DO)` event on the first beat, presumably because in its original context the line was rising. However, after adding this note with no idea of its original context, it is reinterpreted as a `DO` event in the bottom level, and placed immediately before a `(SI MI)` event. This rising step followed by two downward leaps of a diminished fourth and a major third is awkward and uncharacteristic of Bach.

The solution to this is problem to "smooth" the bass line using a Markov method, where each two consecutive events are compared against consecutive events in the Bach corpus and the bass line reset to the exact notes that appear in the local context of the Bach data. Then, any two consecutive bass events in the output appear somewhere in Bach's works as movement of the bass line. This same example appears smoothed in LEUCHTET-V.1.e (p. 53), where the second downward leap has been removed, and the line is acceptable.

Appendix A

Completing Four-Part Harmonizations

While adding bass lines to a melody is a worthy task, I wanted to find out if a statistical method could be applied successfully to complete four-part harmonization. This program is a work in progress, and it does make mistakes. However, it often achieves results comparable or superior to the work of beginning music theory students, despite its shortcomings.

First, the program harmonizes the bass line and a complete four-part cadence, creating a framework for the harmonization. It then goes backward from the cadence in a manner similar to that described in Chapter 5.

Each new chord is completed based on the bass-soprano pair and the four-part harmony in the following one or two already harmonized chords. The program searches for bi- or tri-4VC's in the data set whose last one or two members are equivalent to the following one or two chords. If the bass and soprano are equivalent to those in the first member of the bi- or tri-4VC, its inner voices are inserted into the harmonization. Applications of the weaker

rules for event equivalency listed in Chapter 4 helped to give the program greater flexibility. However, this also led the program to make voice-leading errors, including the forbidden parallel octaves and fifths. I made no attempt to smooth the results or check for errors; this was merely an experiment to determine whether a Markov model similar to that used for bass lines could also accomplish the task of completing harmony.

Examples of this method appear at the end of Appendix B.

Appendix B

Musical Examples

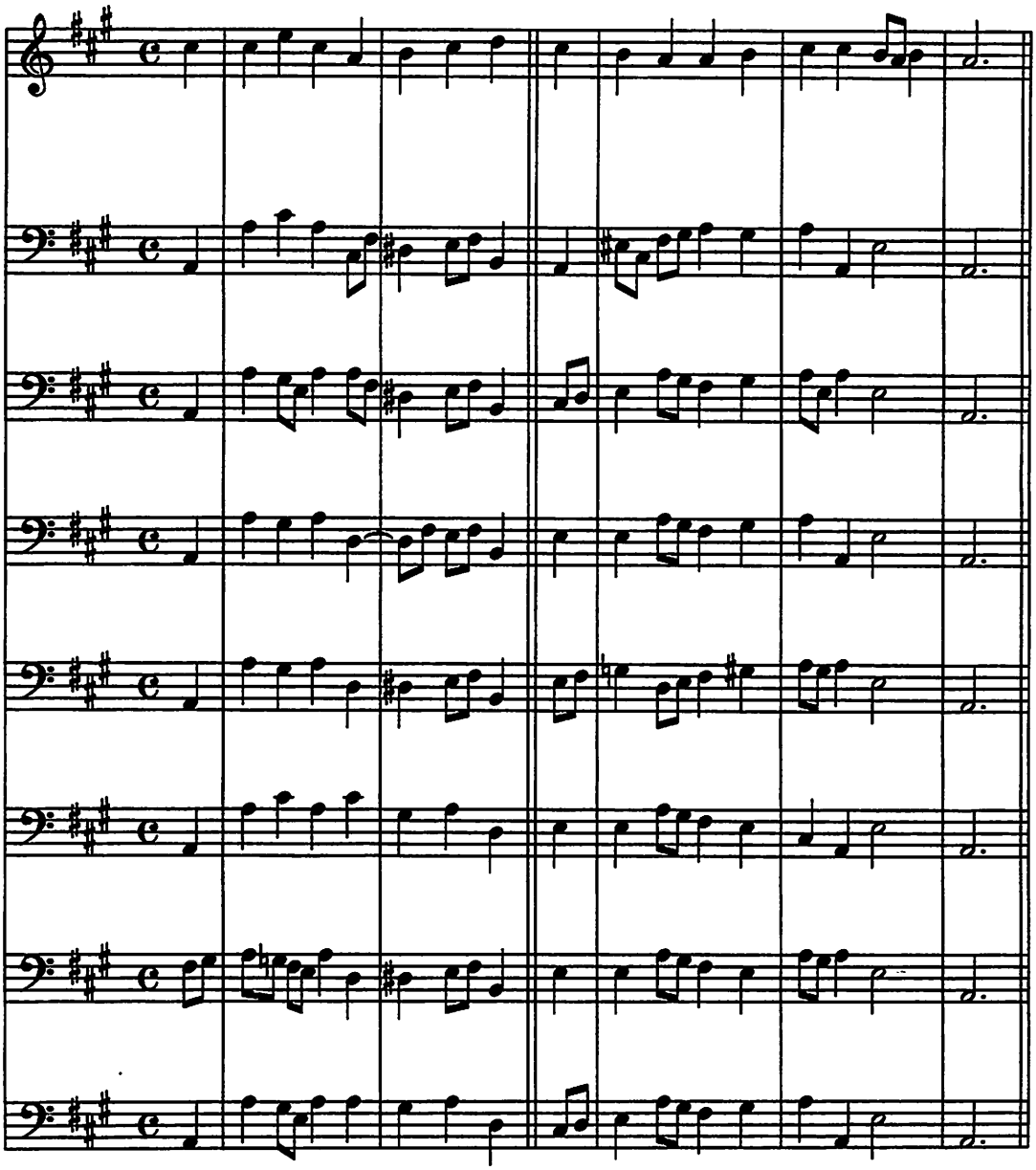
This section contains the musical examples referred to in the text of the thesis.

Each example is named by a keyword from the text of the chorale tune harmonized, the method of harmonization used, the phrase of the tune harmonized, and a letter indicating the particular harmonization for that phrase. For example, LEUCHTET-IV.3.b is the second harmonization the computer generated for the third phrase of "Wie schön *leuchtet* der Morgenstern," using Method IV (The Top-Down Approach with metric information).

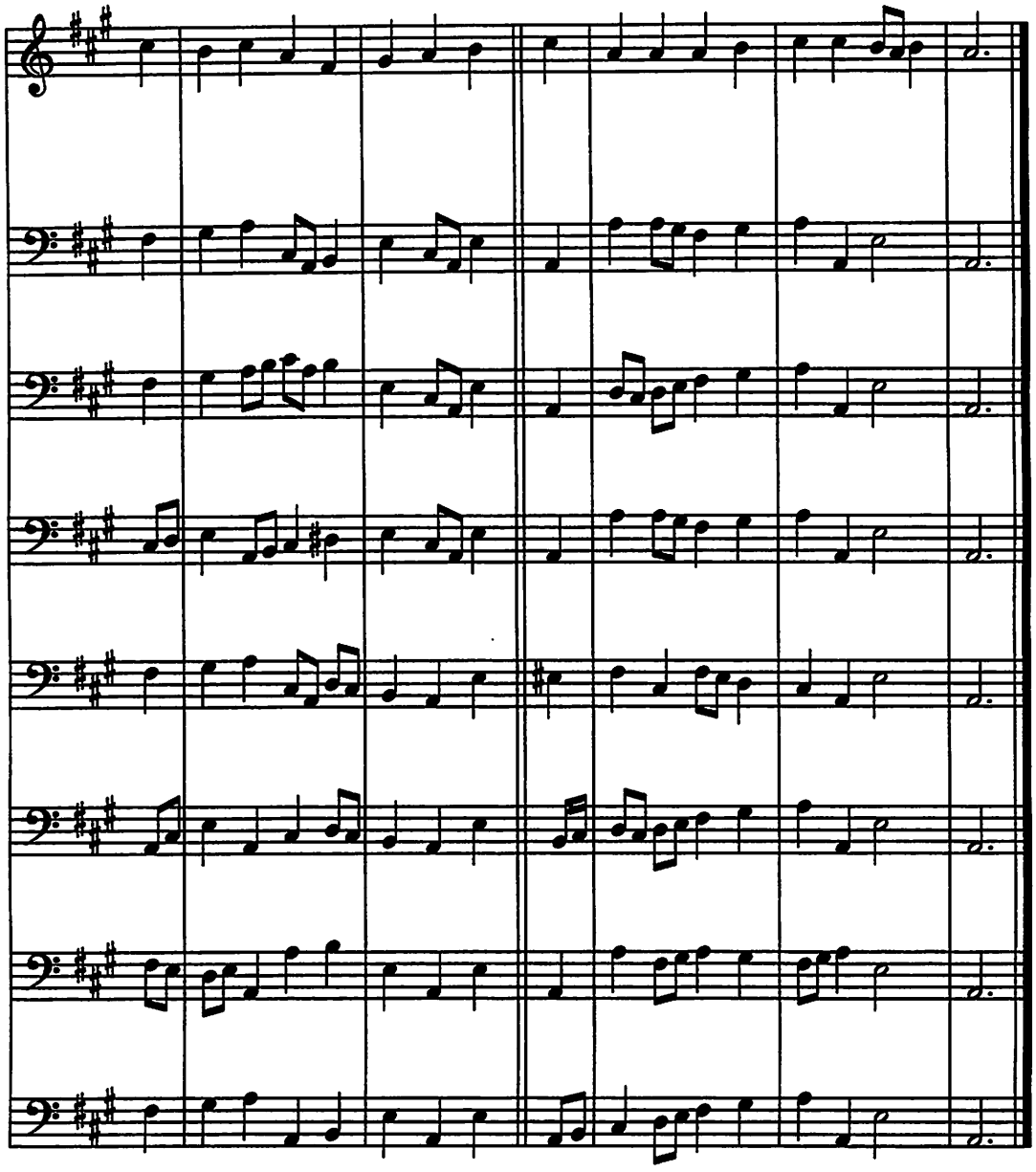
The first set of examples, BLEIB-I.1.a through LEUCHTET-V.5.g, consists of bass lines the program generated for various phrases. Each phrase, or melodic unit, appears at the top of a column. The bass lines the program generated for it appear below. Phrases are separated by double barlines.

The final set of examples consists of complete four-part harmonizations the program generated for the same phrases in the BLEIB and LEUCHTET examples. Some attempt has been made to organize these phrases so that the same bass lines for a given melody appear in the same column, but this is not a universal rule.

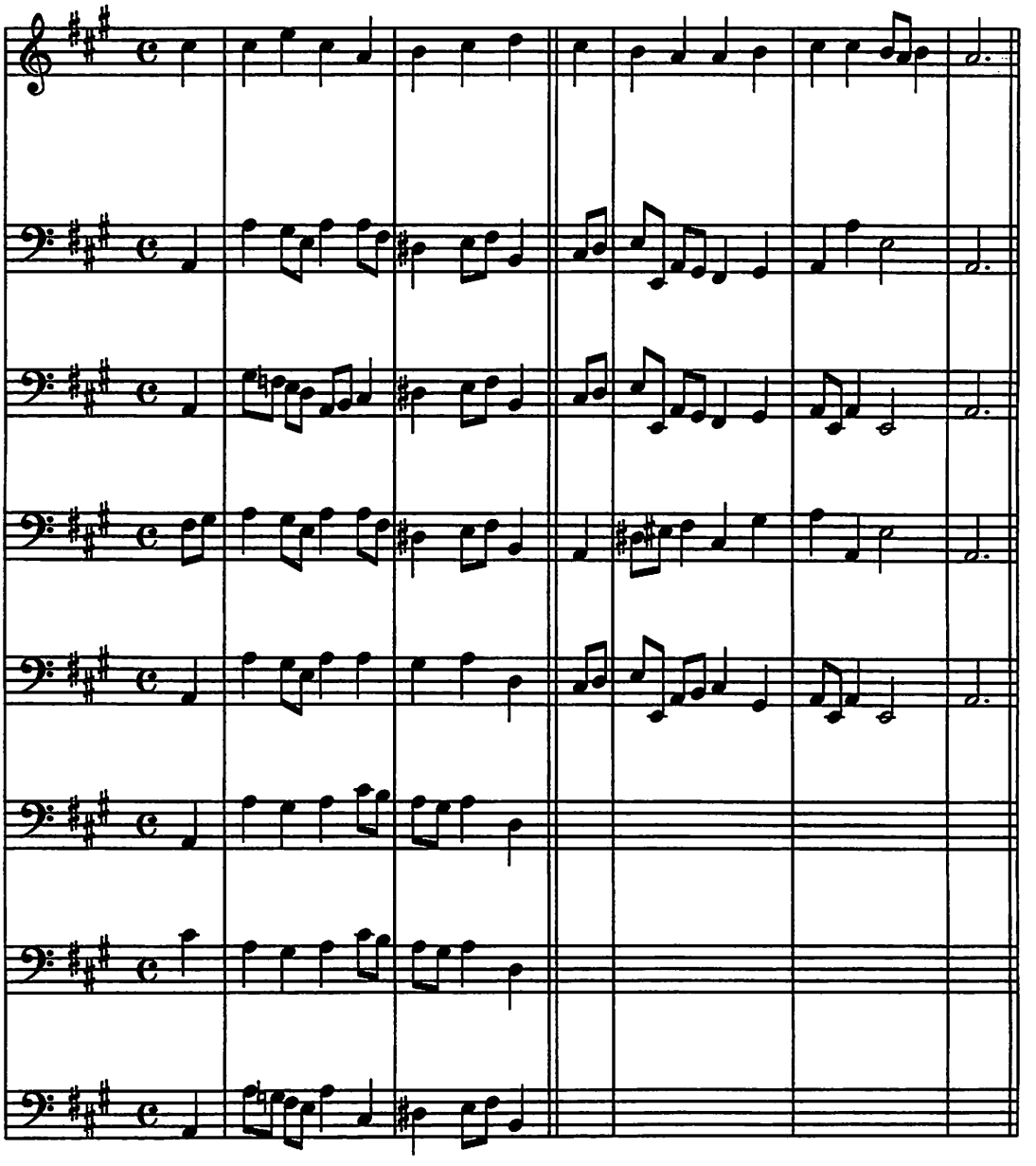
Melody: "Ach bleib bei uns, Herr Jesu Christ" (BLEIB)
Method I: The Simple Approach
Phrases 1 and 2
Left Column: Examples BLEIB-I.1.a to BLEIB-I.1.g
Right Column: Examples BLEIB-I.2.a to BLEIB-I.2.g



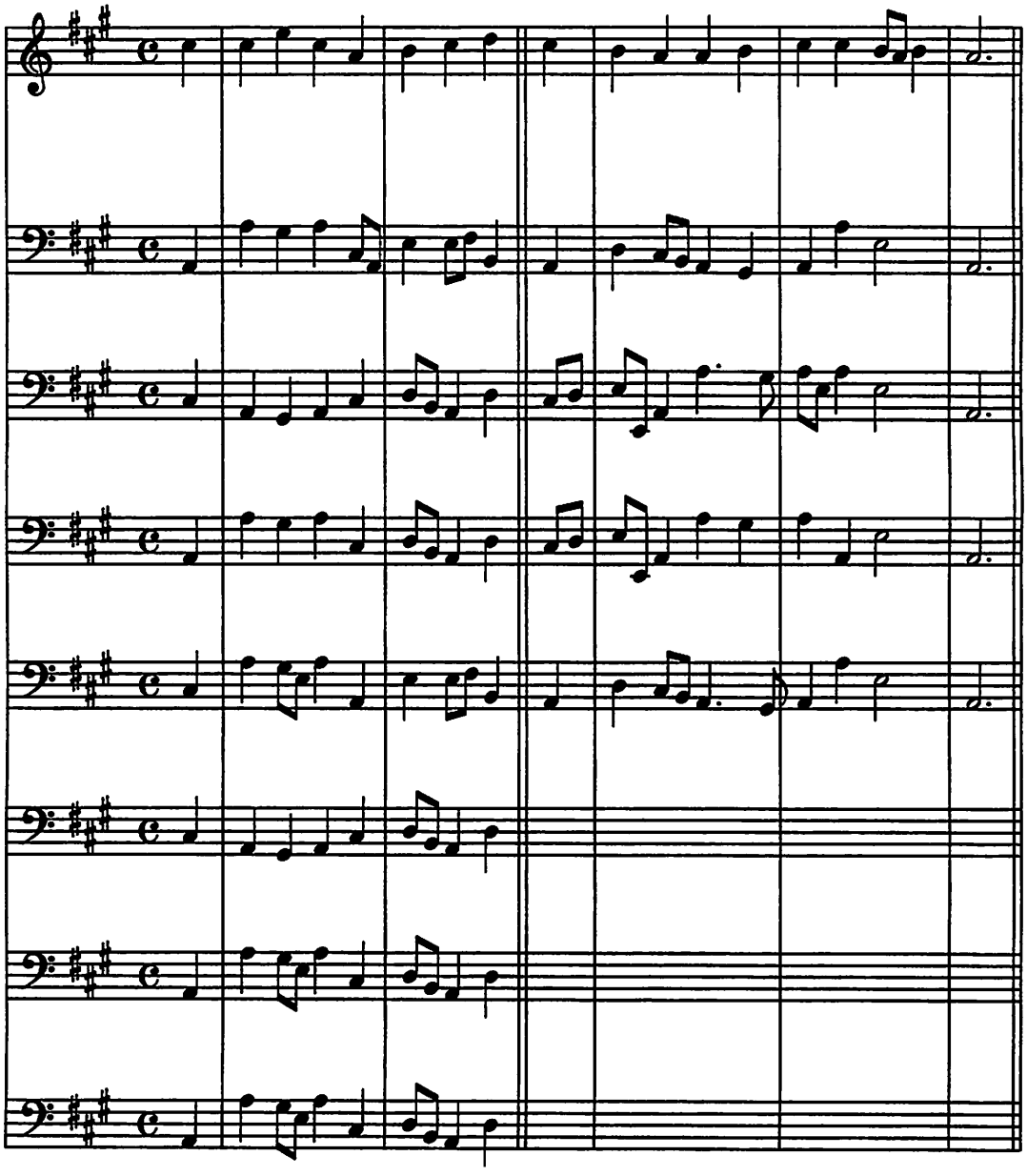
Melody: "Ach bleib bei uns, Herr Jesu Christ" (BLEIB)
Method I: The Simple Approach
Phrases 3 and 4
Left Column: Examples BLEIB-I.3.a to BLEIB-I.3.g
Right Column: Examples BLEIB-I.4.a to BLEIB-I.4.g



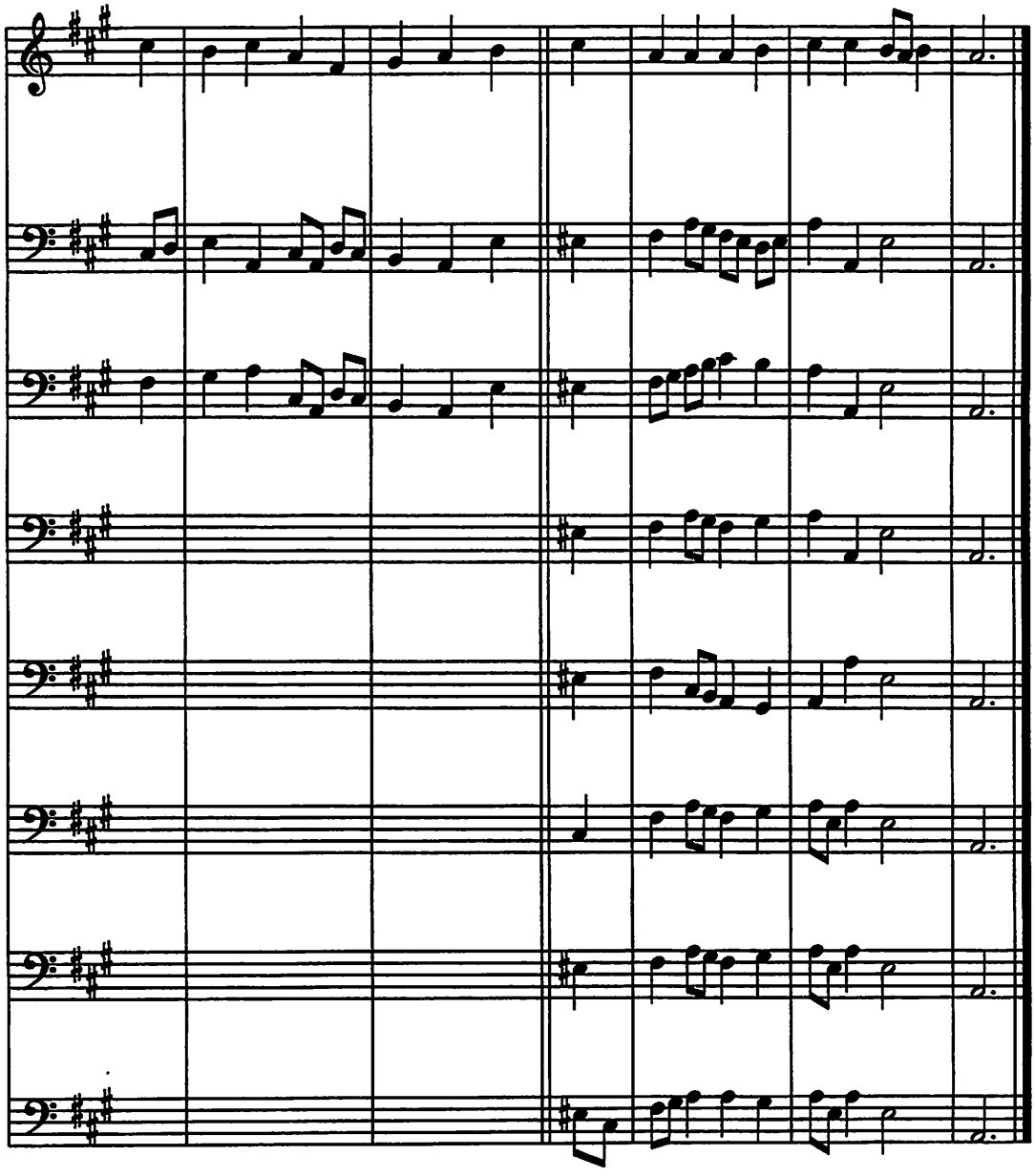
Melody: "Ach bleib bei uns, Herr Jesu Christ" (BLEIB)
Method II: The Simple Approach using metric information
Phrases 1 and 2
Left Column: Examples BLEIB-II.1.a to BLEIB-II.1.g
Right Column: Examples BLEIB-II.2.a to BLEIB-II.2.d



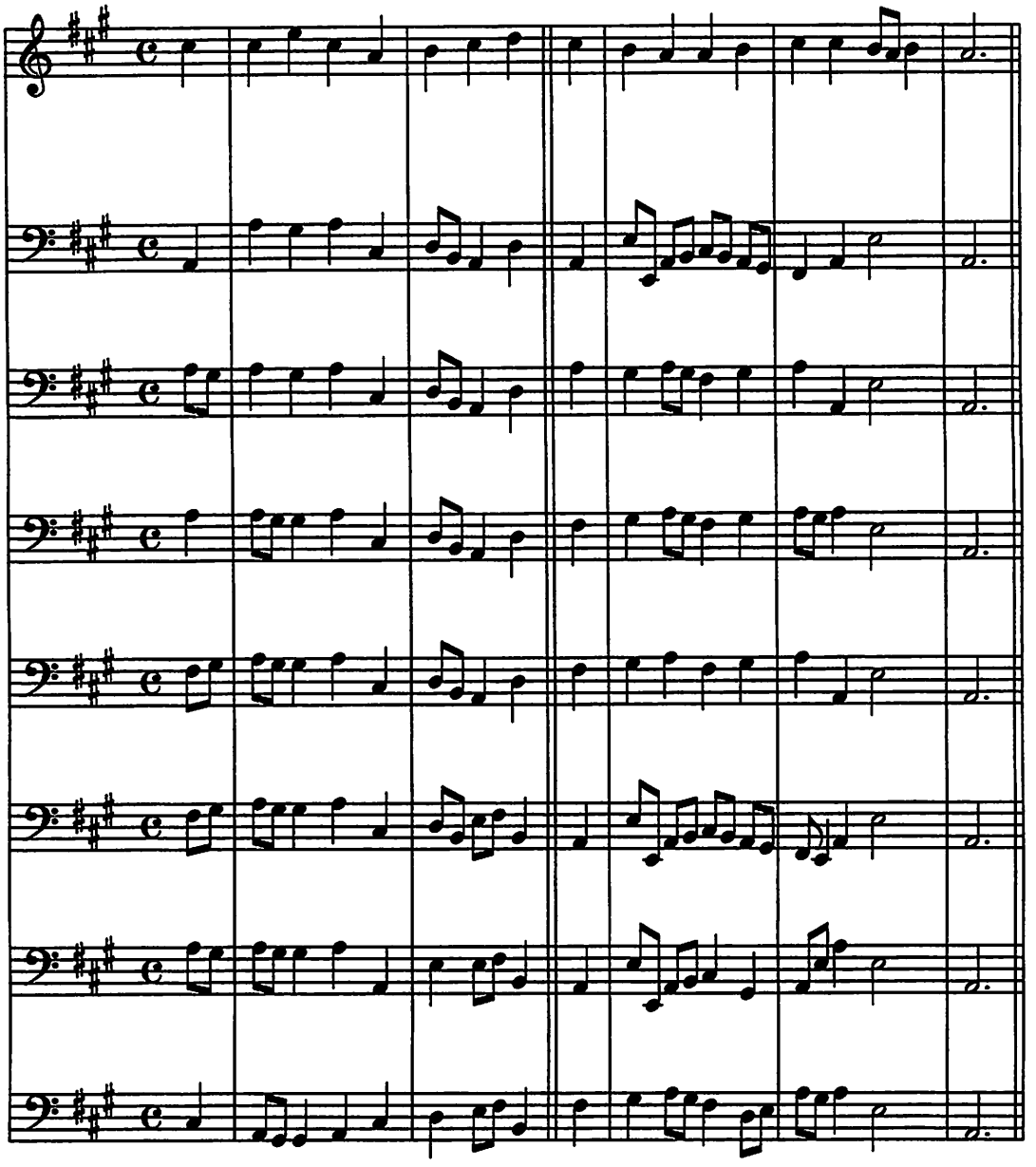
Melody: "Ach bleib bei uns, Herr Jesu Christ" (BLEIB)
Method III: The Top-Down Approach
Phrases 1 and 2
Left Column: Examples BLEIB-III.1.a to BLEIB-III.1.g
Right Column: Examples BLEIB-III.2.a to BLEIB-III.2.g



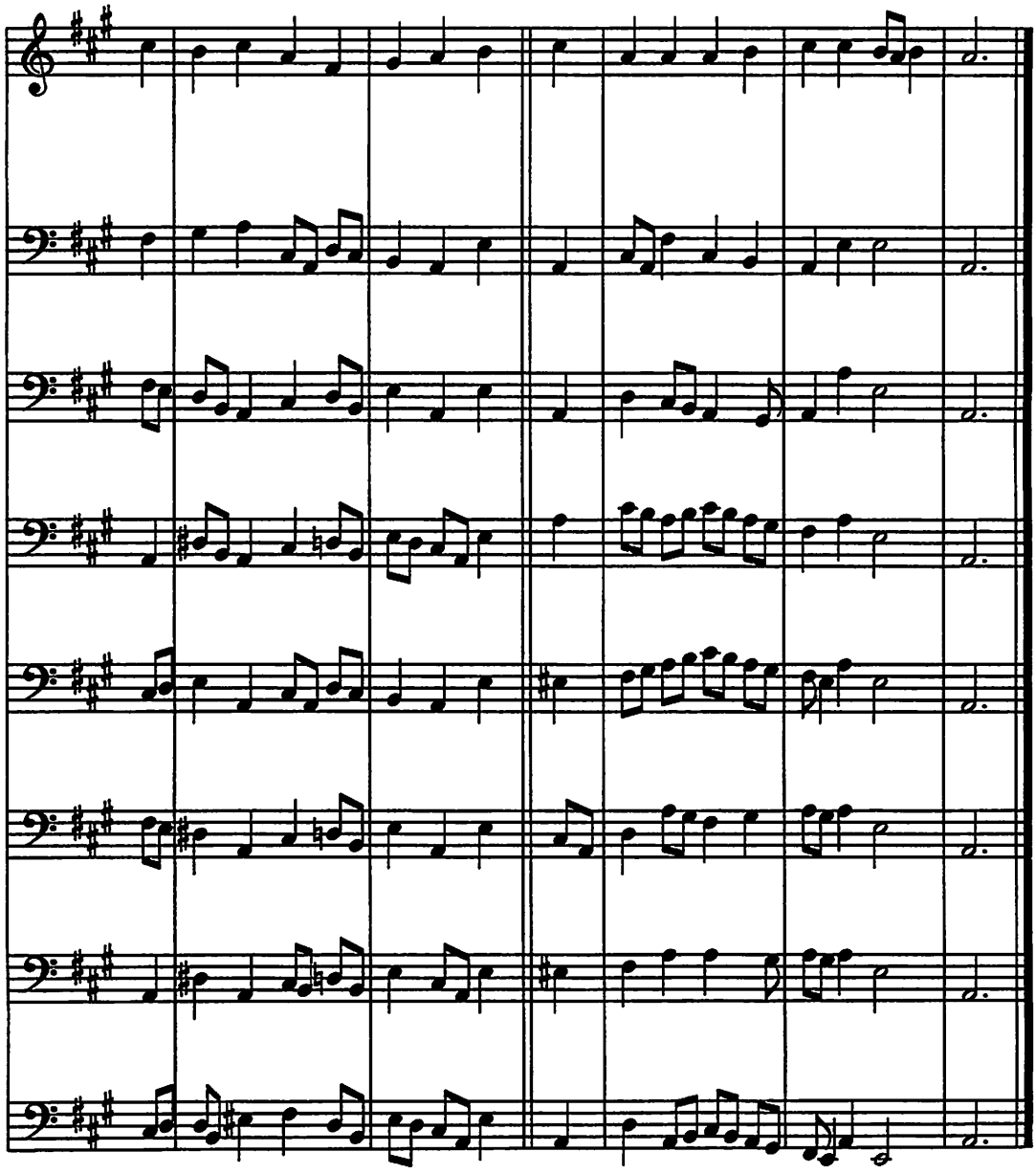
Melody: "Ach bleib bei uns, Herr Jesu Christ" (BLEIB)
Method III: The Top-Down Approach
Phrases 3 and 4
Left Column: Examples BLEIB-III.3.a to BLEIB-III.3.g
Right Column: Examples BLEIB-III.4.a to BLEIB-III.4.g



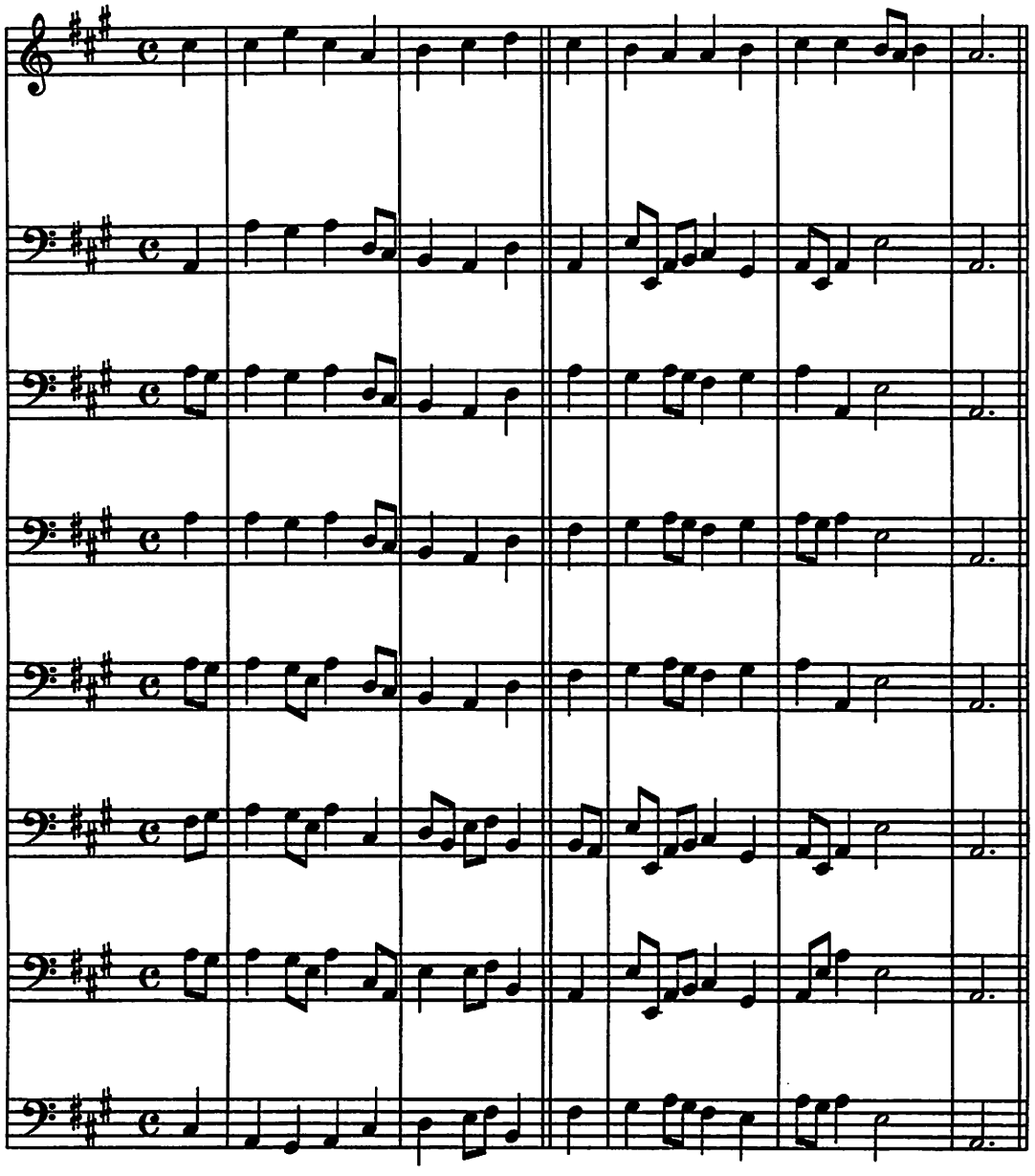
Melody: "Ach bleib bei uns, Herr Jesu Christ" (BLEIB)
Method IV: The Top-Down Approach using metric information
Phrases 1 and 2
Left Column: Examples BLEIB-IV.1.a to BLEIB-IV.1.g
Right Column: Examples BLEIB-IV.2.a to BLEIB-IV.2.g



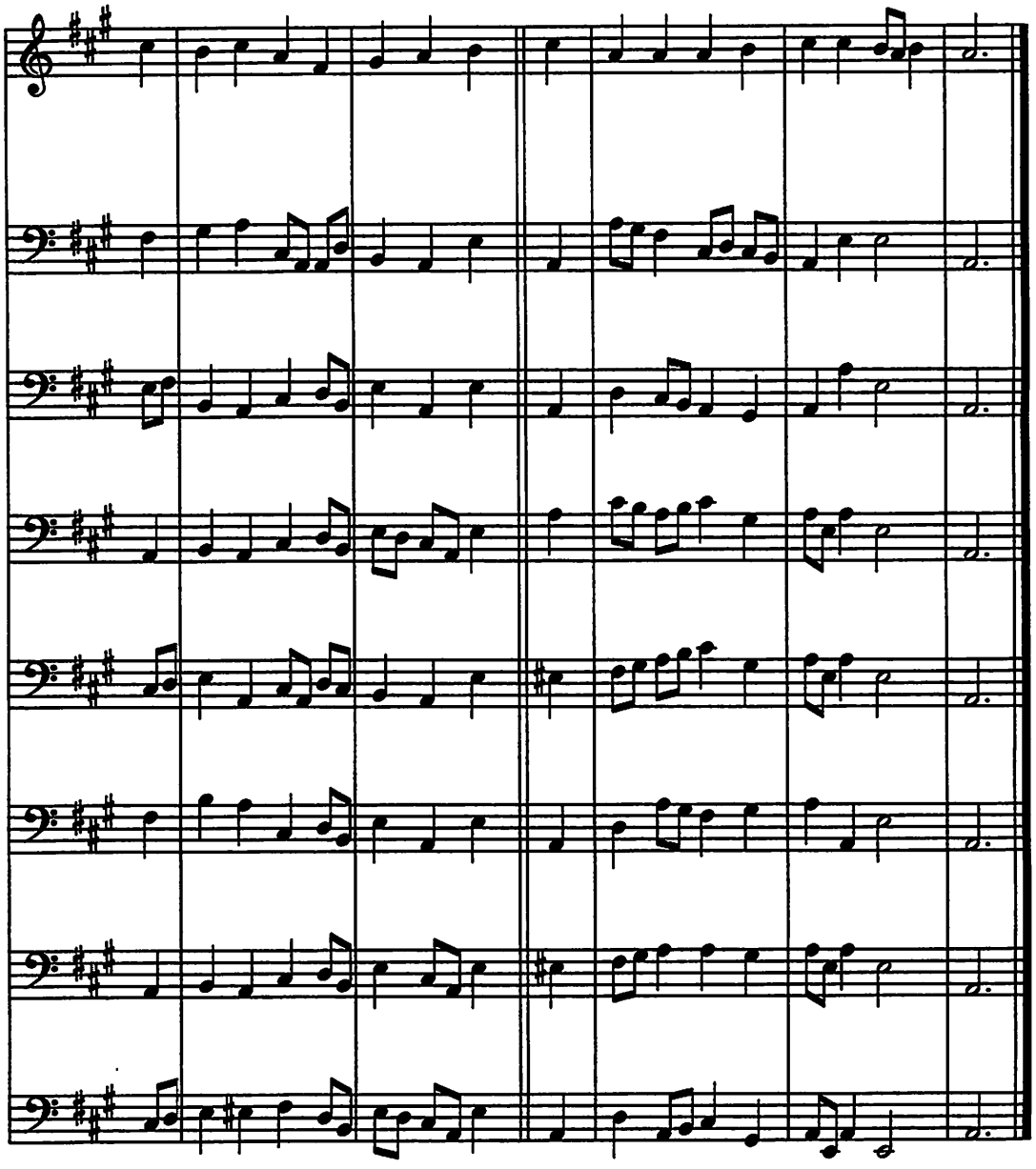
Melody: "Ach bleib bei uns, Herr Jesu Christ" (BLEIB)
Method IV: The Top-Down Approach using metric information
Phrases 3 and 4
Left Column: Examples BLEIB-IV.3.a to BLEIB-IV.3.g
Right Column: Examples BLEIB-IV.4.a to BLEIB-IV.4.g



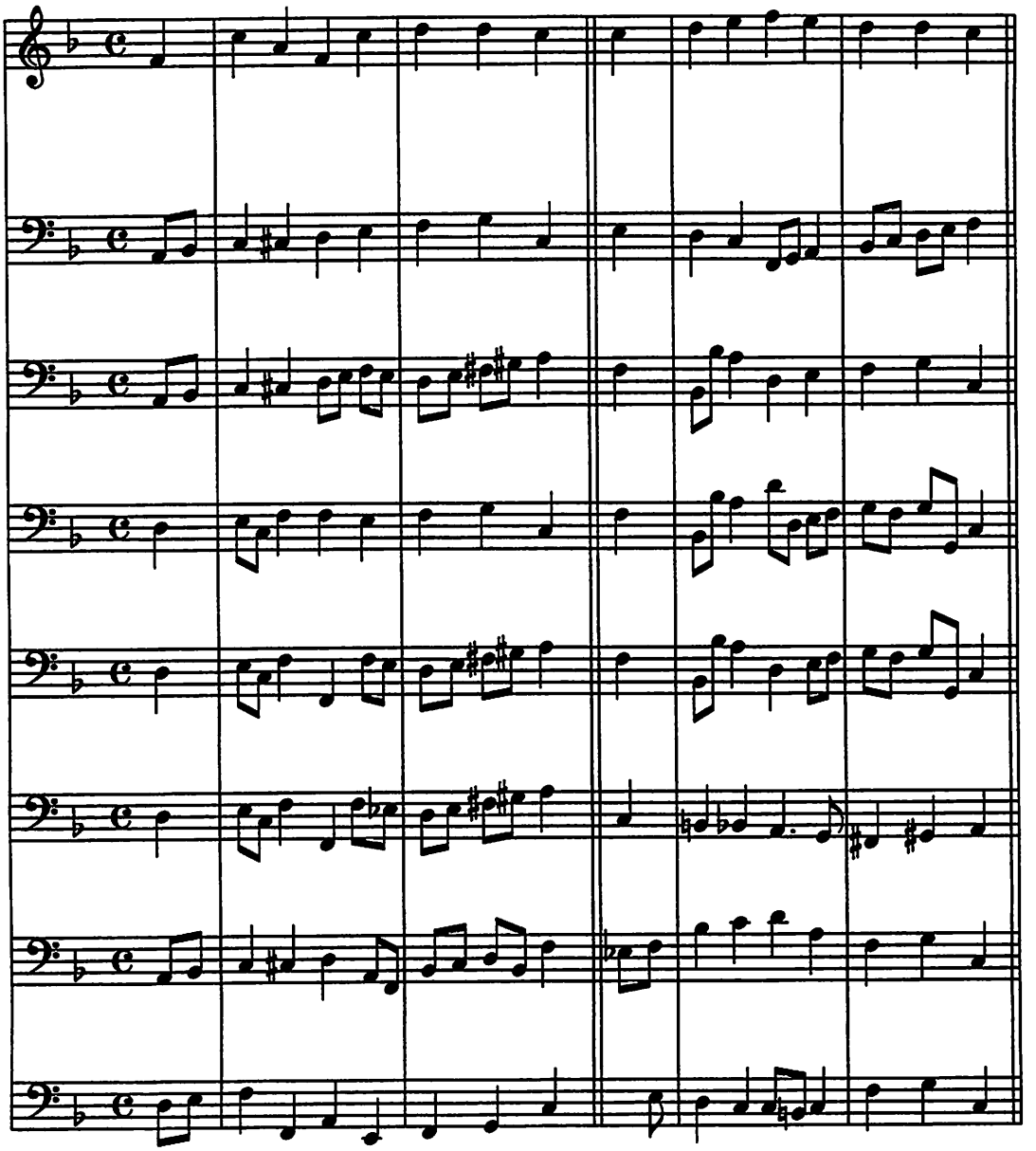
Melody: "Ach bleib bei uns, Herr Jesu Christ" (BLEIB)
Method V: Smoothed Top-Down Approach using metric information
Phrases 1 and 2
Left Column: Examples BLEIB-V.1.a to BLEIB-V.1.g
Right Column: Examples BLEIB-V.2.a to BLEIB-V.2.g



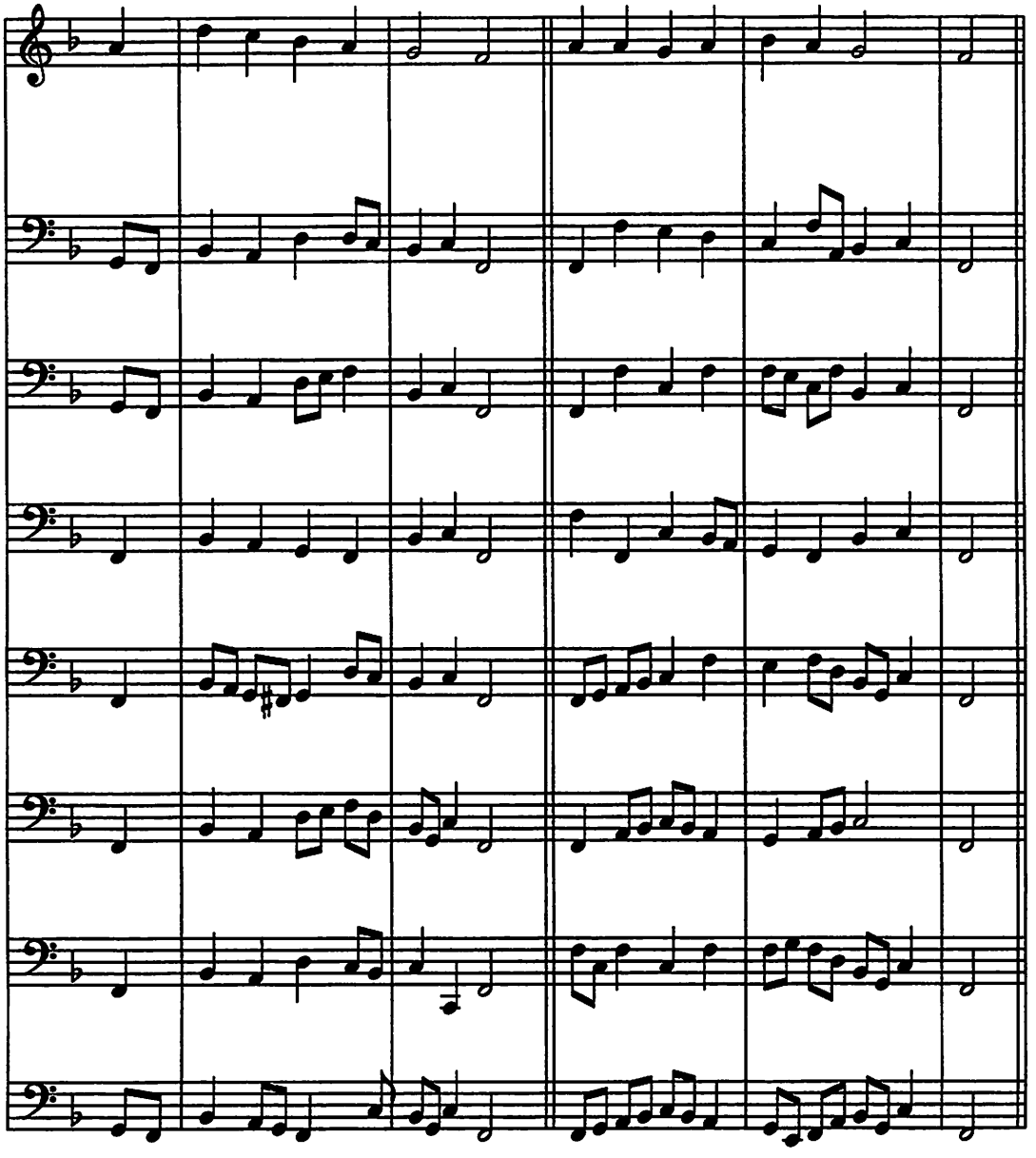
Melody: "Ach bleib bei uns, Herr Jesu Christ" (BLEIB)
Method V: Smoothed Top-Down Approach using metric information
Phrases 3 and 4
Left Column: Examples BLEIB-V.3.a to BLEIB-V.3.g
Right Column: Examples BLEIB-V.4.a to BLEIB-V.4.g



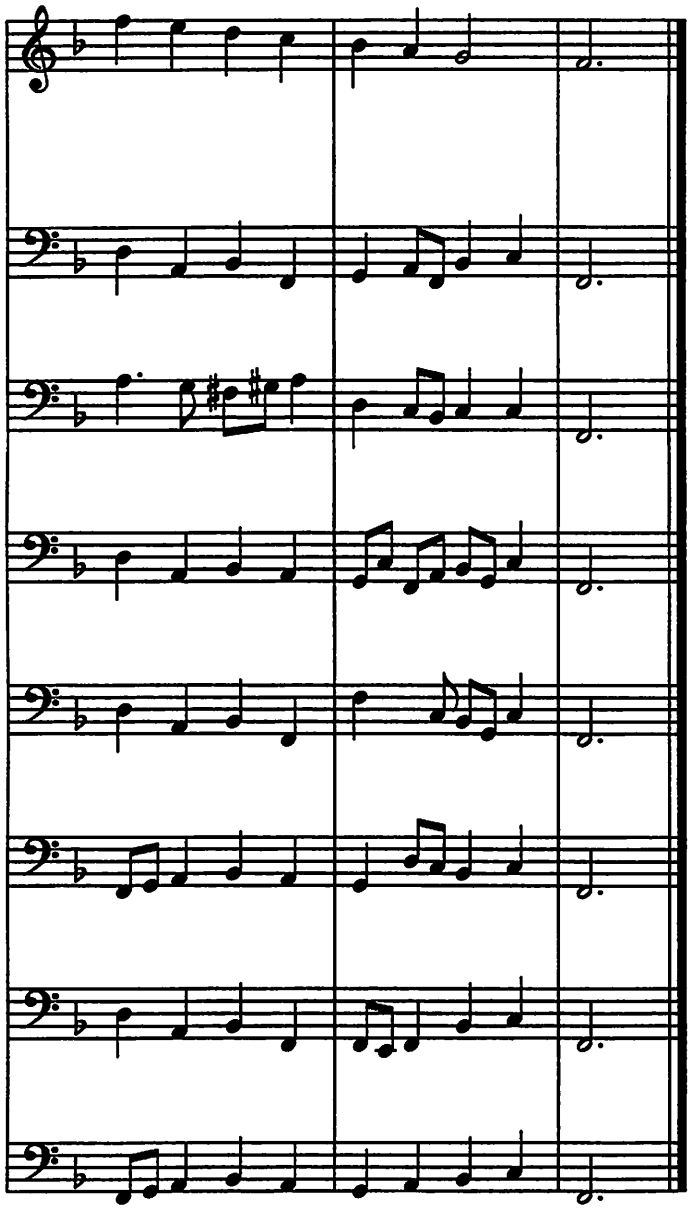
Melody: "Wie schön leuchtet der Morgenstern" (LEUCHTET)
Method I: The Simple Approach
Phrases 1 and 2
Left Column: Examples LEUCHTET-I.1.a to LEUCHTET-I.1.g
Right Column: Examples LEUCHTET-I.2.a to LEUCHTET-I.2.g



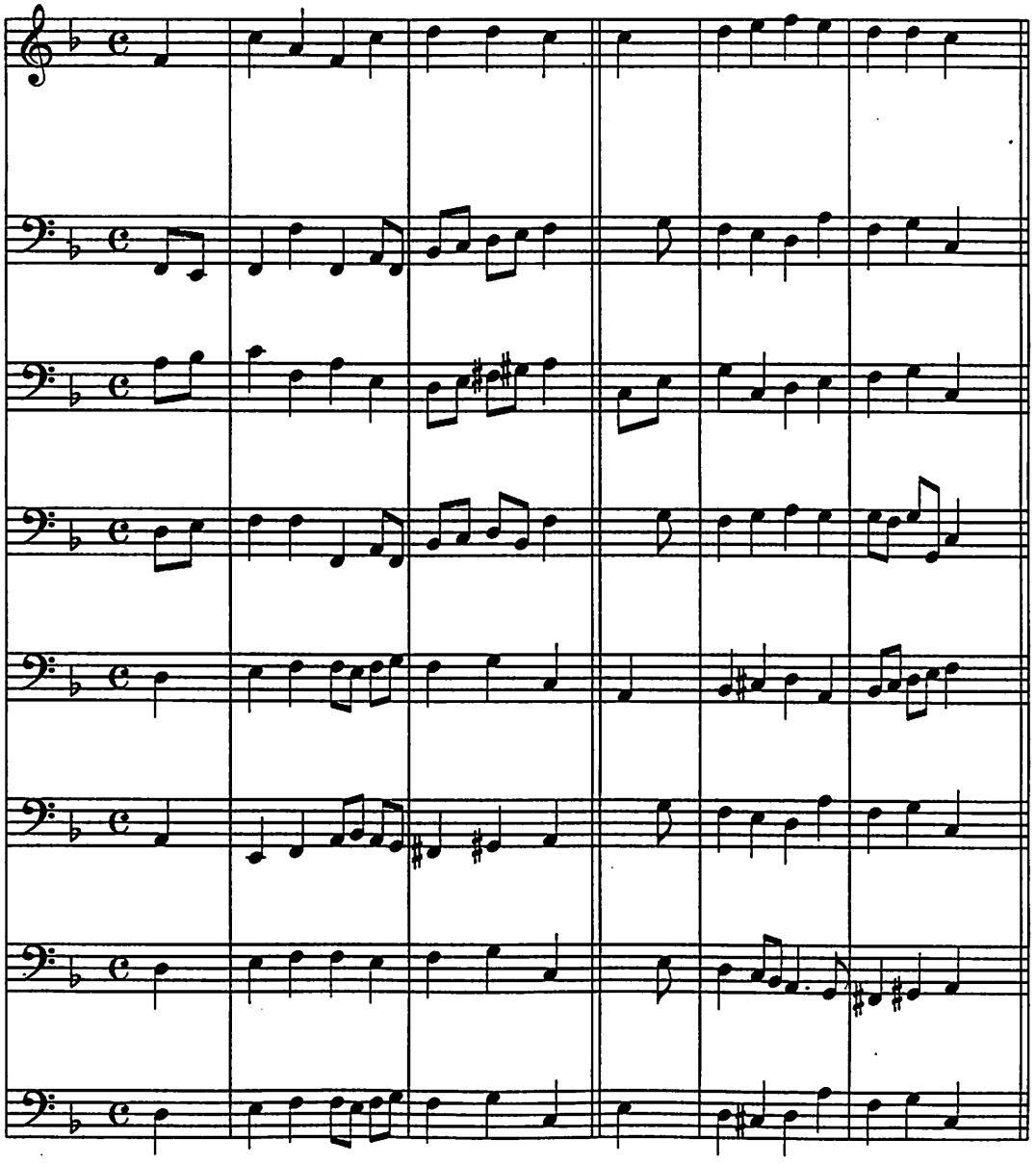
Melody: "Wie schön leuchtet der Morgenstern" (LEUCHTET)
Method I: The Simple Approach
Phrases 3 and 4
Left Column: Examples LEUCHTET-I.3.a to LEUCHTET-I.3.g
Right Column: Examples LEUCHTET-I.4.a to LEUCHTET-I.4.g



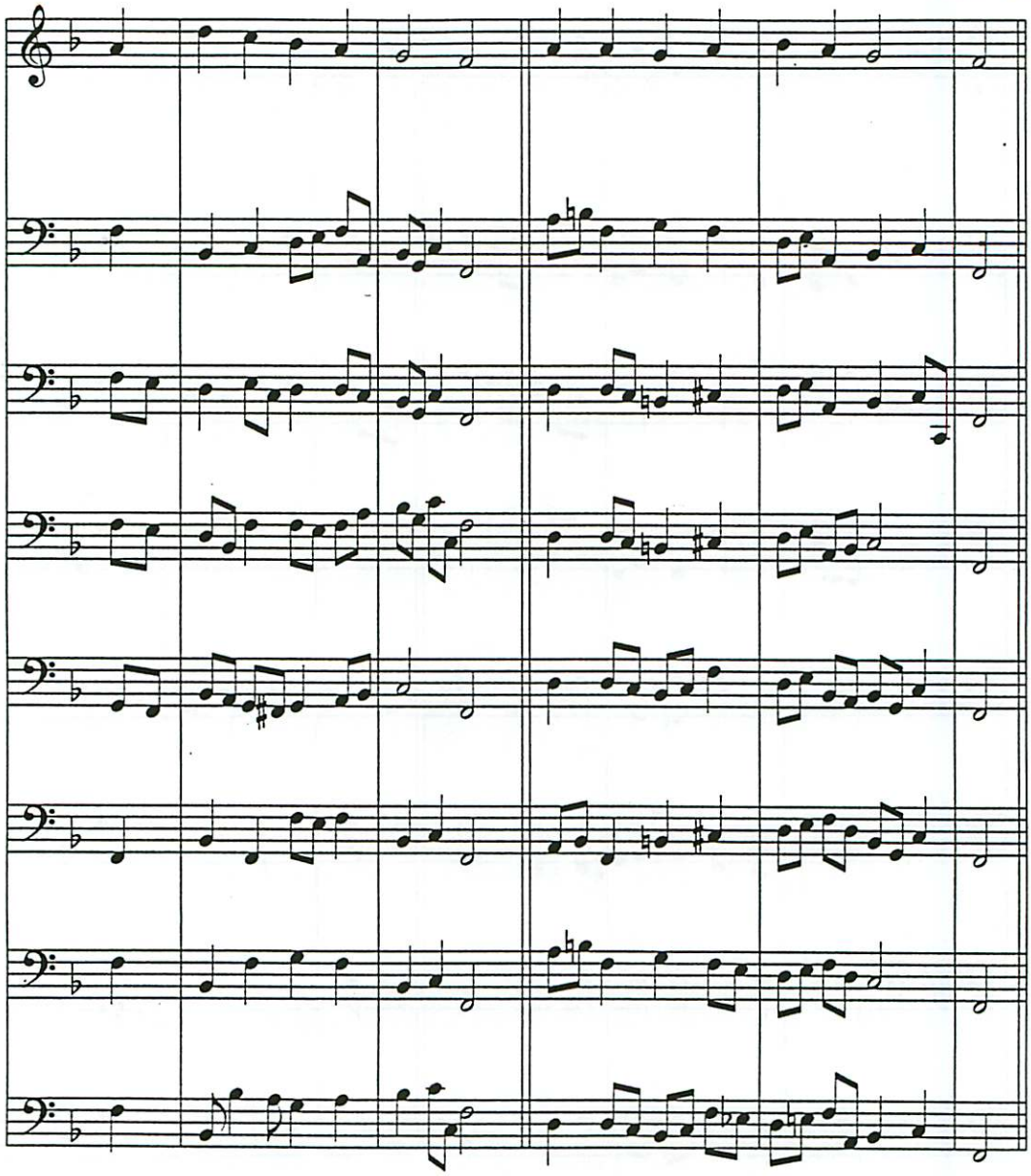
Melody: "Wie schön leuchtet der Morgenstern" (LEUCHTET)
Method I: The Simple Approach
Phrase 5
Examples LEUCHTET-I.5.a to LEUCHTET-I.5.g



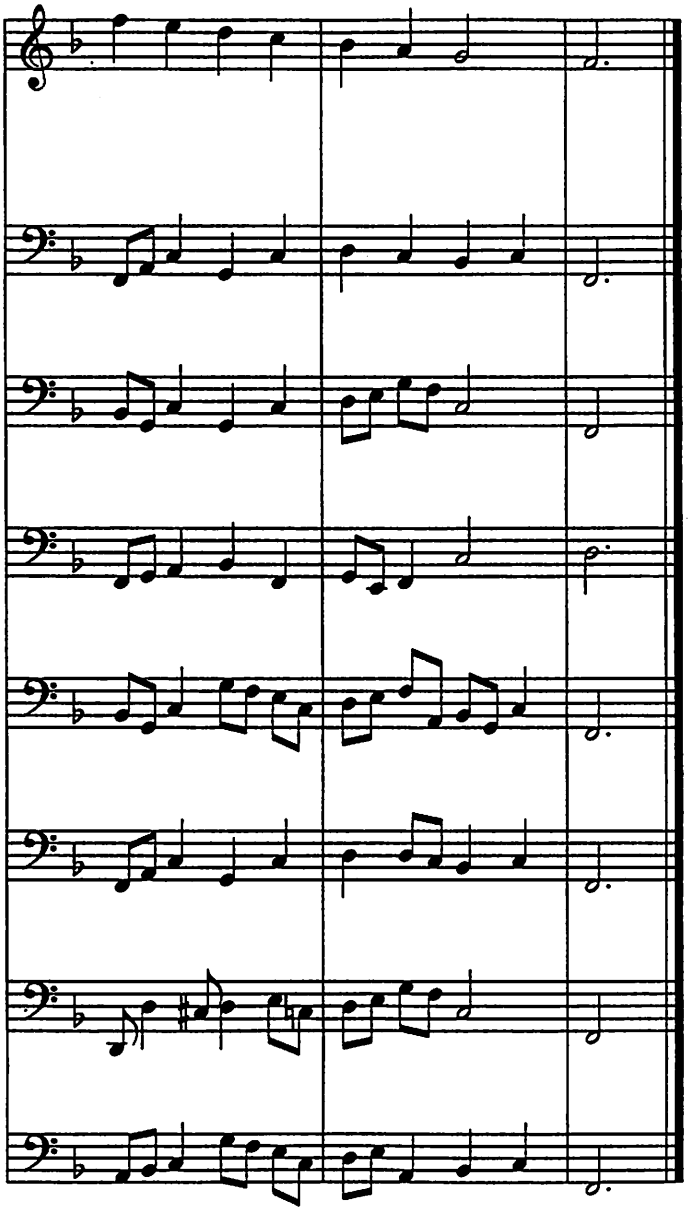
Melody: "Wie schön leuchtet der Morgenstern" (LEUCHTET)
Method III: The Top-Down Approach
Phrases 1 and 2
Left Column: Examples LEUCHTET-III.1.a to LEUCHTET-III.1.g
Right Column: Examples LEUCHTET-III.2.a to LEUCHTET-III.2.g



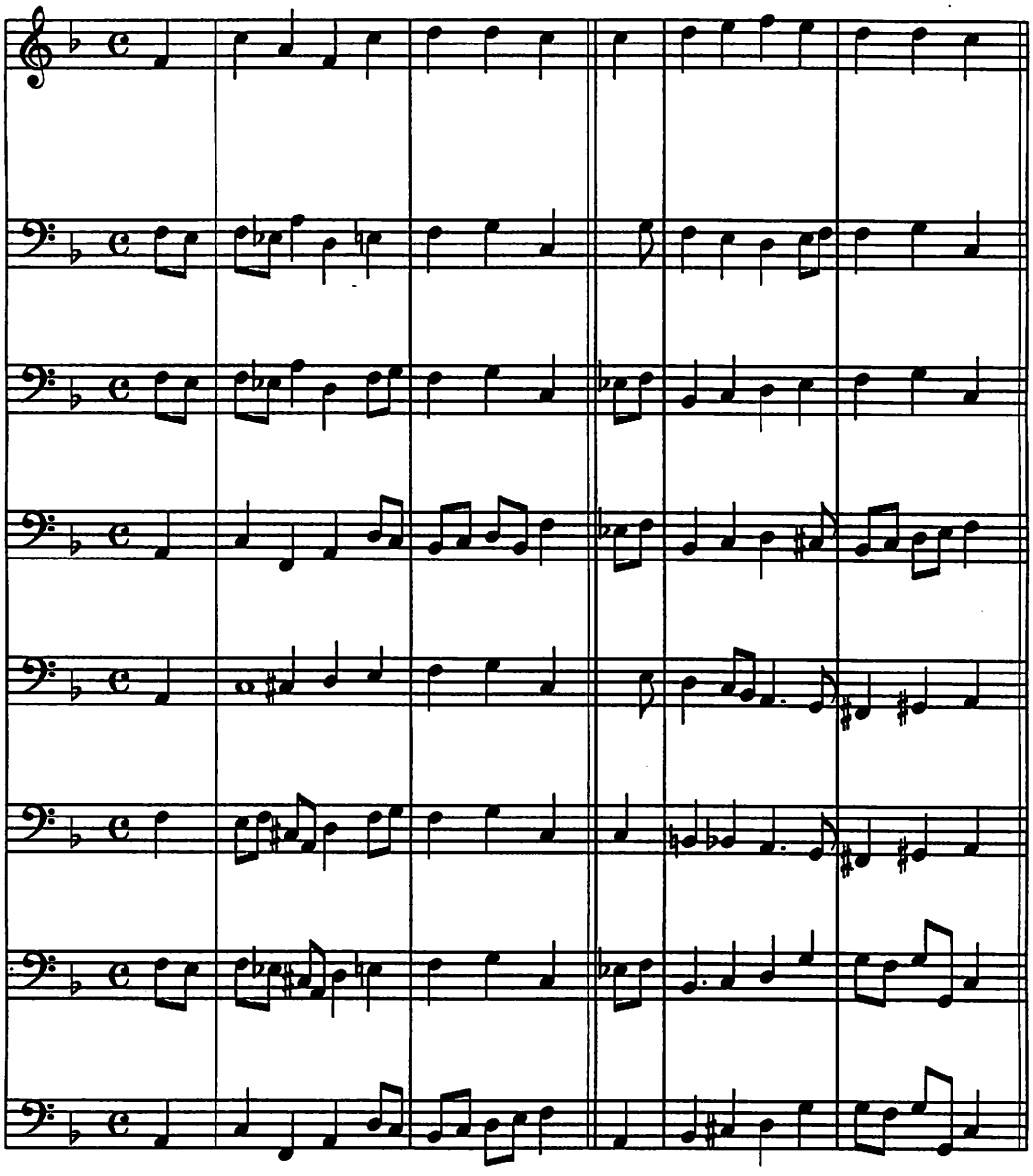
Melody: "Wie schön leuchtet der Morgenstern" (LEUCHTET)
Method III: The Top-Down Approach
Phrases 3 and 4
Left Column: Examples LEUCHTET-III.3.a to LEUCHTET-III.3.g
Right Column: Examples LEUCHTET-III.4.a to LEUCHTET-III.4.g



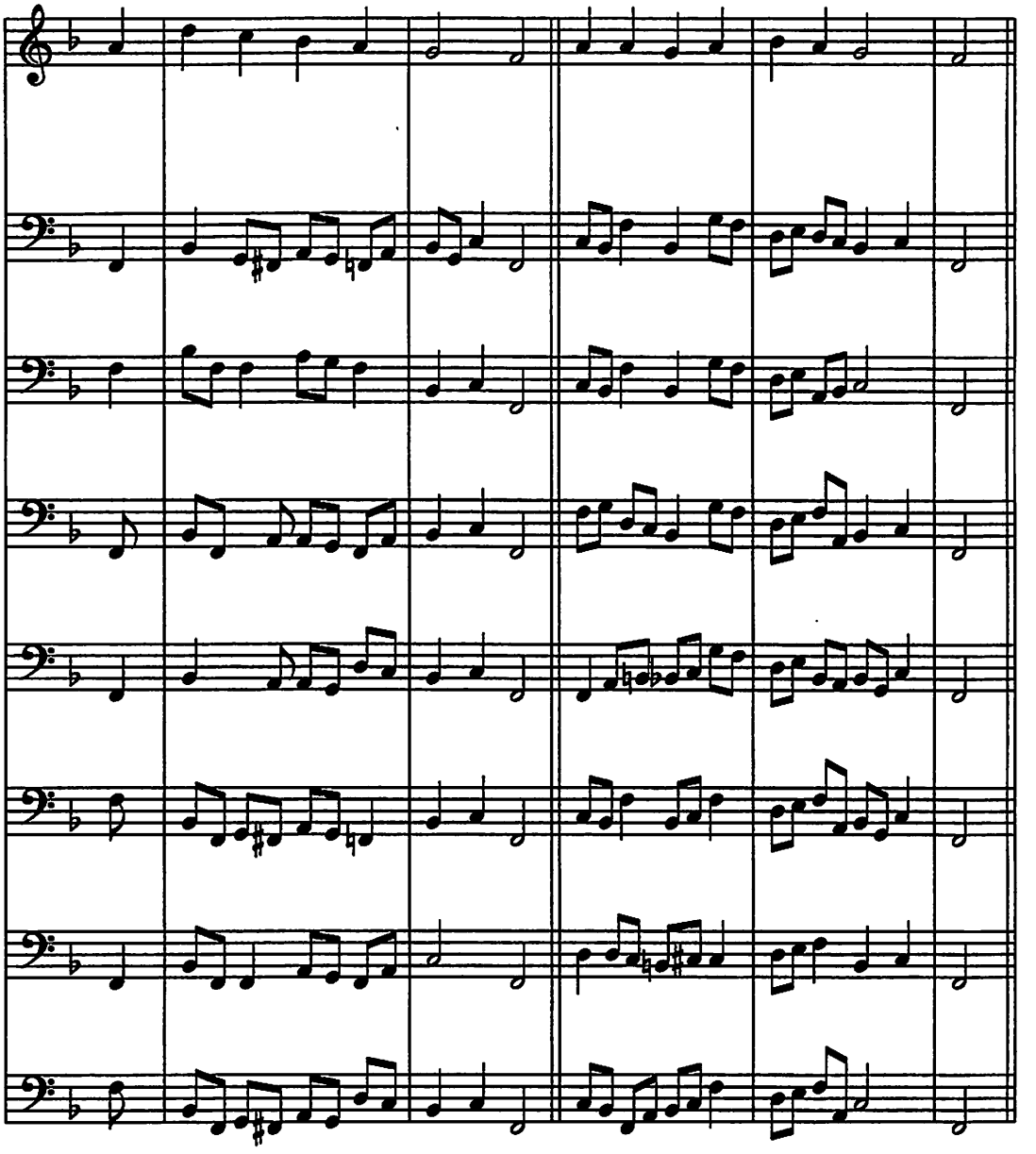
Melody: "Wie schön leuchtet der Morgenstern" (LEUCHTET)
Method III: The Top-Down Approach
Phrase 5
Examples LEUCHTET-III.5.a to LEUCHTET-III.5.g



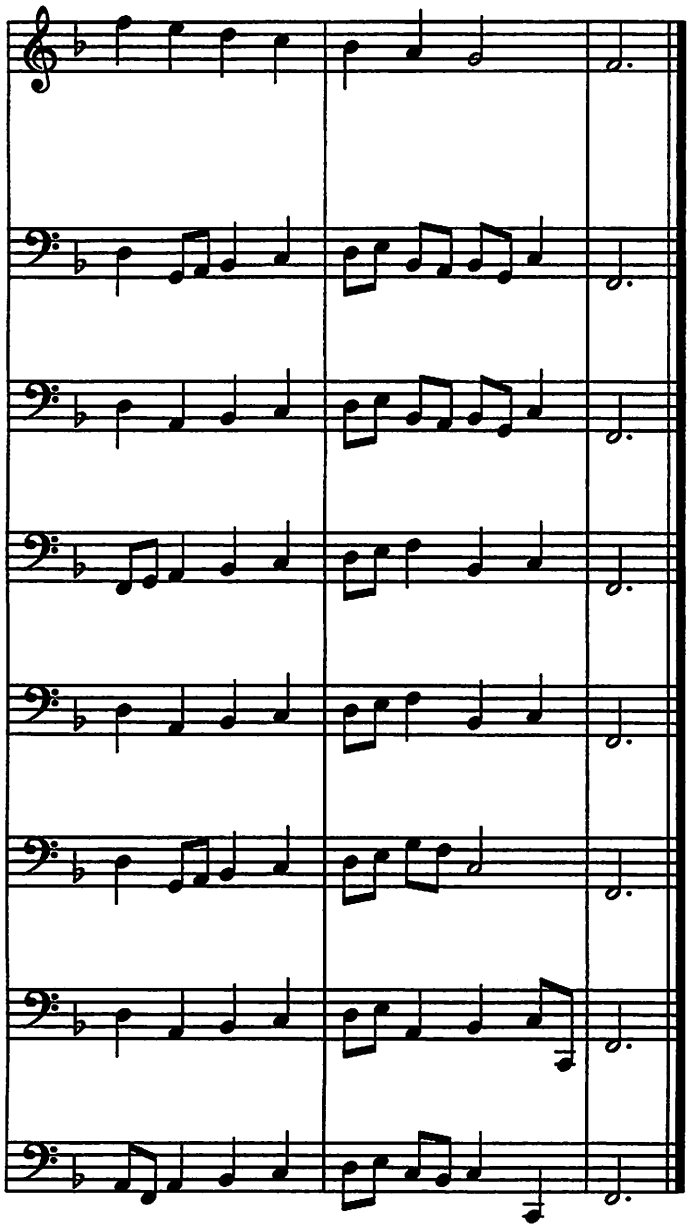
Melody: "Wie schön leuchtet der Morgenstern" (LEUCHTET)
Method IV: The Top-Down Approach using metric information
Phrases 1 and 2
Left Column: Examples LEUCHTET-IV.1.a to LEUCHTET-IV.1.g
Right Column: Examples LEUCHTET-IV.2.a to LEUCHTET-IV.2.g



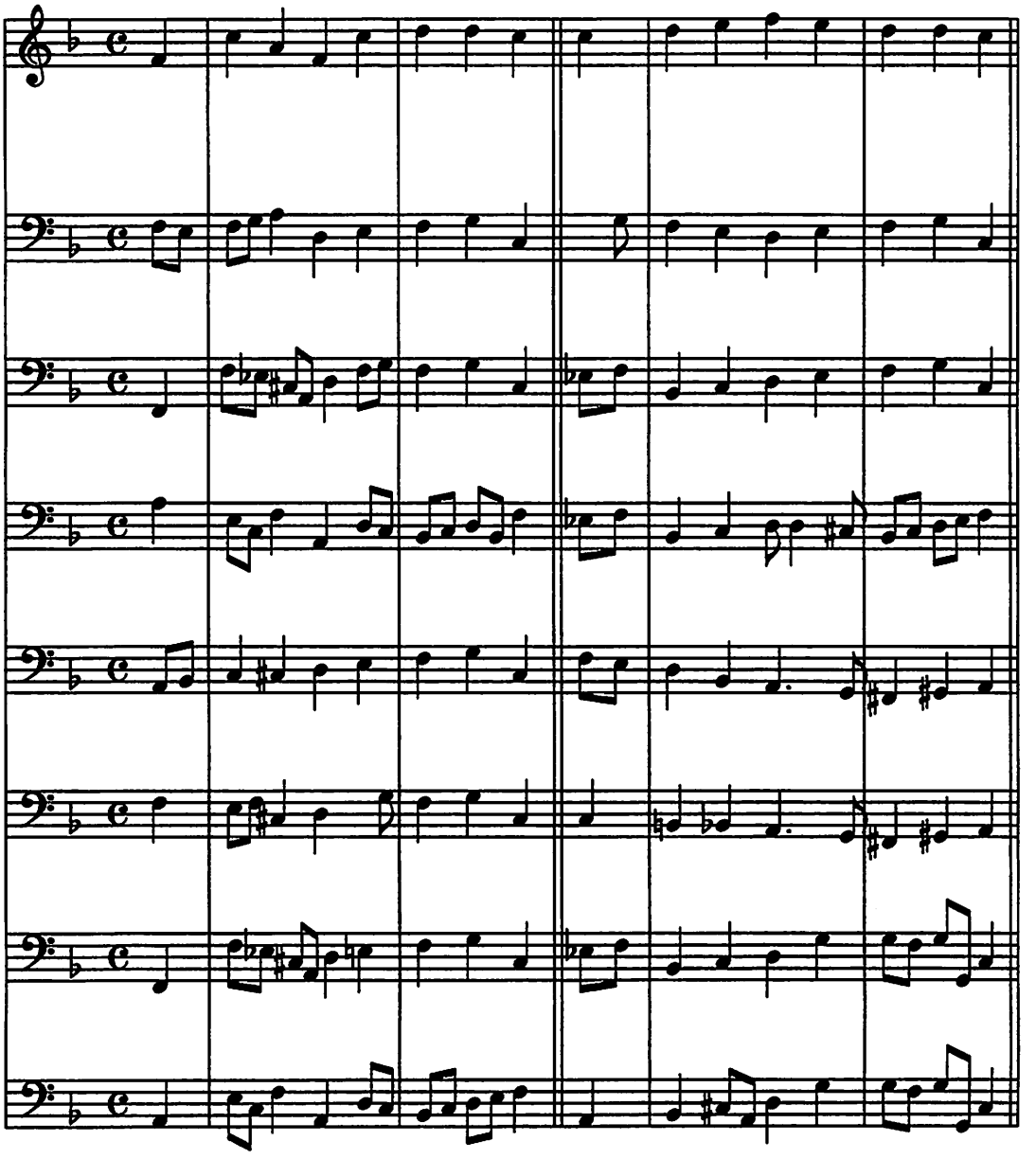
Melody: "Wie schön leuchtet der Morgenstern" (LEUCHTET)
Method IV: The Top-Down Approach using metric information
Phrases 3 and 4
Left Column: Examples LEUCHTET-IV.3.a to LEUCHTET-IV.3.g
Right Column: Examples LEUCHTET-IV.4.a to LEUCHTET-IV.4.g



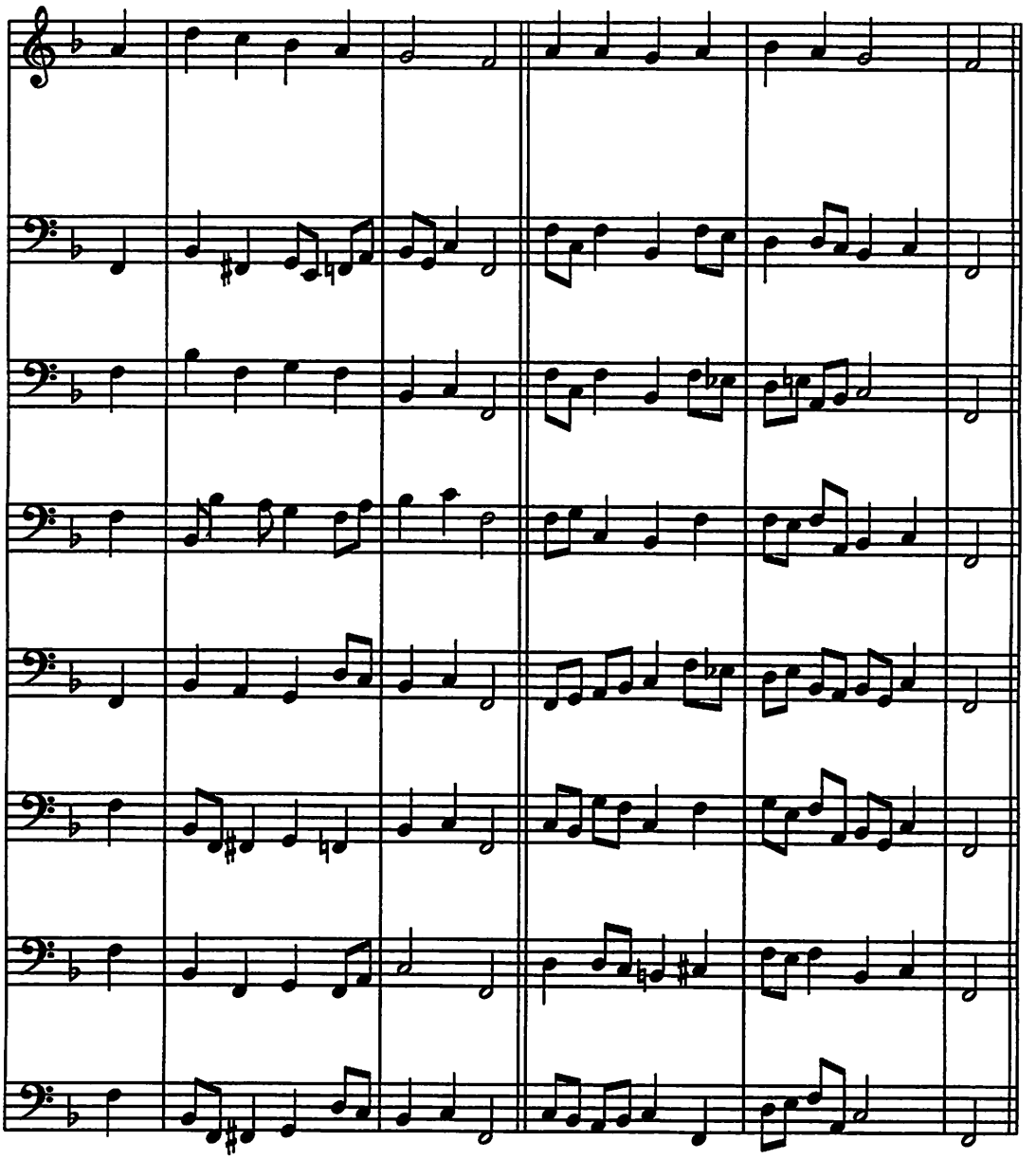
Melody: "Wie schön leuchtet der Morgenstern" (LEUCHTET)
Method IV: The Top-Down Approach using metric information
Phrase 5
Examples LEUCHTET-IV.5.a to LEUCHTET-IV.5.g



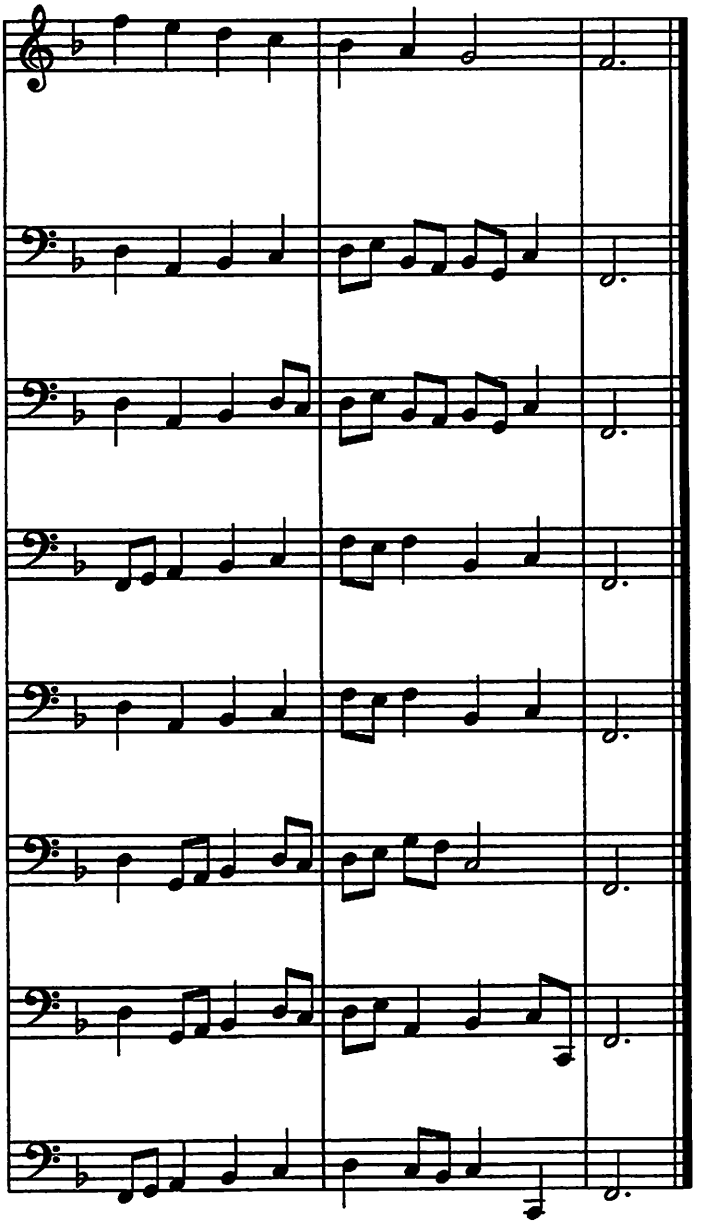
Melody: "Wie schön leuchtet der Morgenstern" (LEUCHTET)
Method V: Smoothed Top-Down Approach using metric information
Phrases 1 and 2
Left Column: Examples LEUCHTET-V.1.a to LEUCHTET-V.1.g
Right Column: Examples LEUCHTET-V.2.a to LEUCHTET-V.2.g



Melody: "Wie schön leuchtet der Morgenstern" (LEUCHTET)
Method V: Smoothed Top-Down Approach using metric information
Phrases 3 and 4
Left Column: Examples LEUCHTET-V.3.a to LEUCHTET-V.3.g
Right Column: Examples LEUCHTET-V.4.a to LEUCHTET-V.4.g

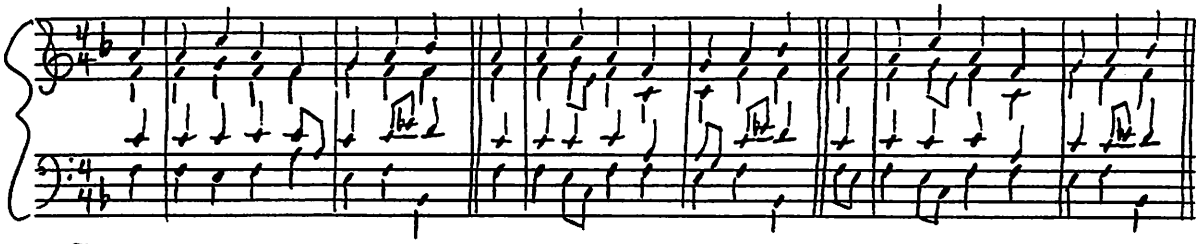


Melody: "Wie schön leuchtet der Morgenstern" (LEUCHTET)
Method V: Smoothed Top-Down Approach using metric information
Phrase 5
Examples LEUCHTET-V.5.a to LEUCHTET-V.5.g

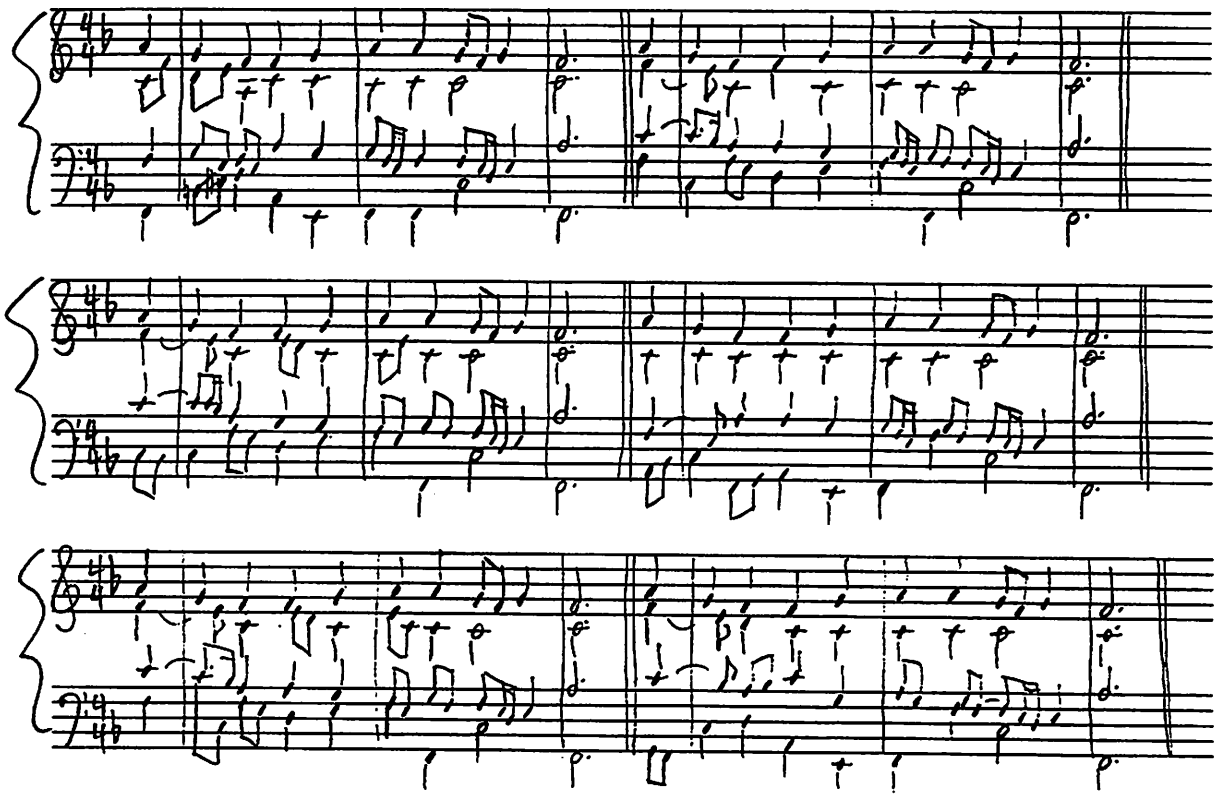


Melody: "Ach bleib bei uns, Herr Jesu Christ" (BLEIB)
Method A: The Simple Approach
Phrases 1 and 2
Examples BLEIB-A.1.a-c and BLEIB-A.2.a-f

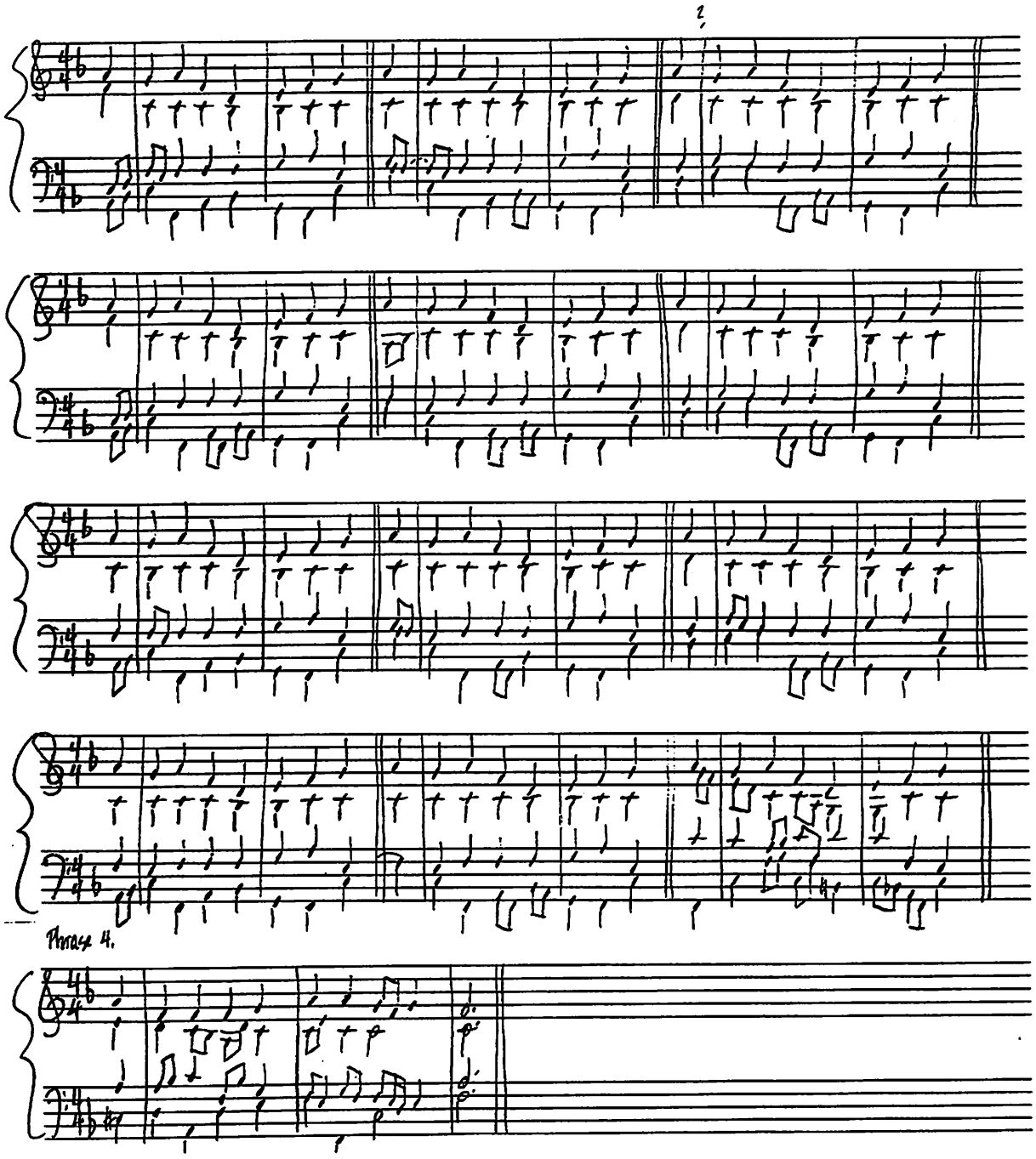
BLEIB PHRASE 1.



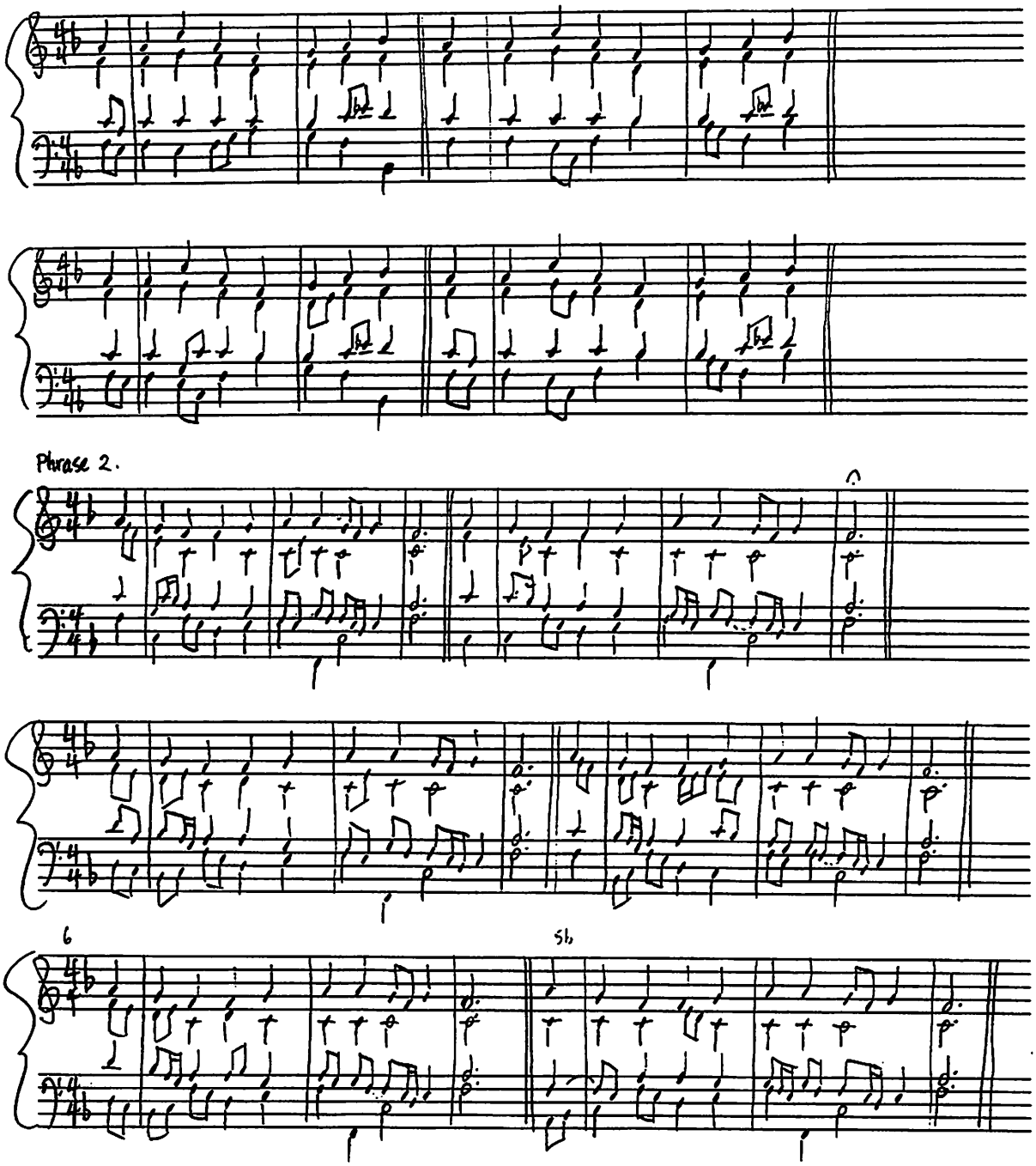
Phrase 2.



Melody: "Ach bleib bei uns, Herr Jesu Christ" (BLEIB)
Method A: The Simple Approach
Phrases 3 and 4
Examples BLEIB-A.3.a-f and BLEIB-A.4.a-d

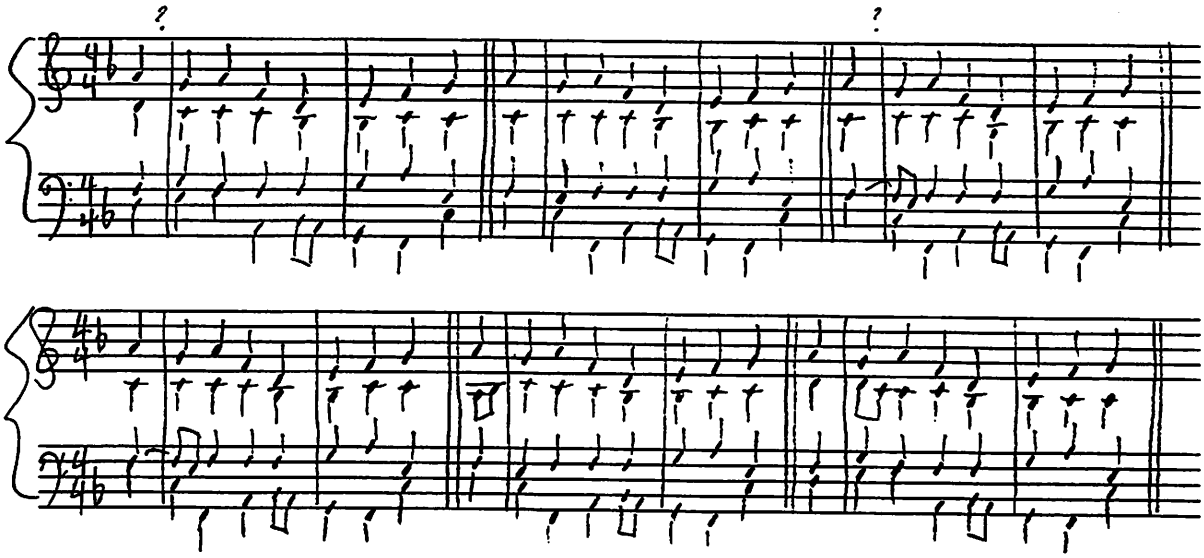


Melody: "Ach bleib bei uns, Herr Jesu Christ" (BLEIB)
Method B: The Top-Down Approach
Phrases 1 and 2
Examples BLEIB-B.1.a-d and BLEIB-B.2.a-f

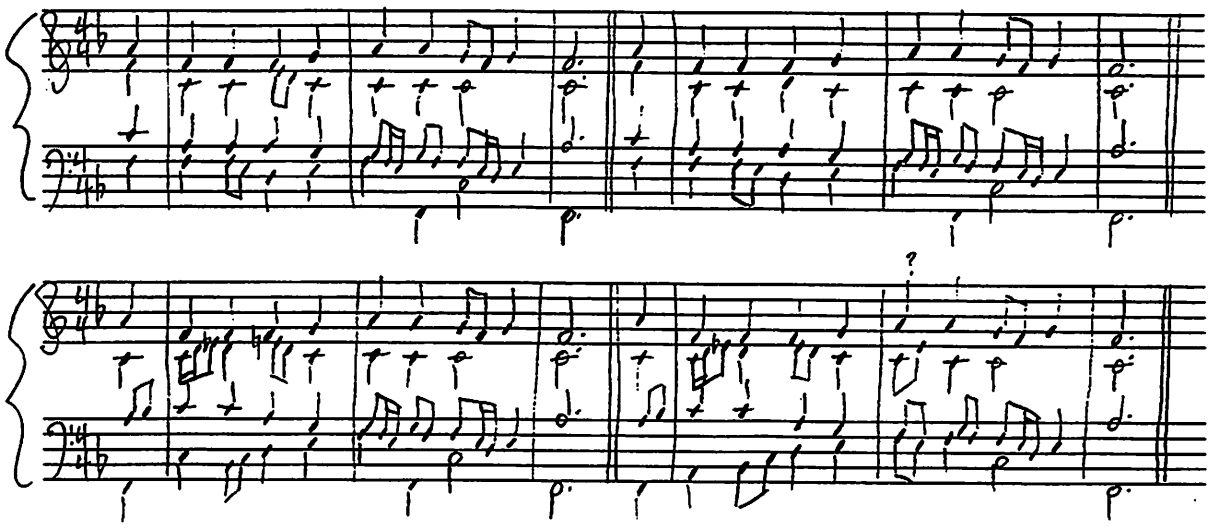


Melody: "Ach bleib bei uns, Herr Jesu Christ" (BLEIB)
Method B: The Top-Down Approach
Phrases 3 and 4
Examples BLEIB-B.3.a-l and BLEIB-B.4.a

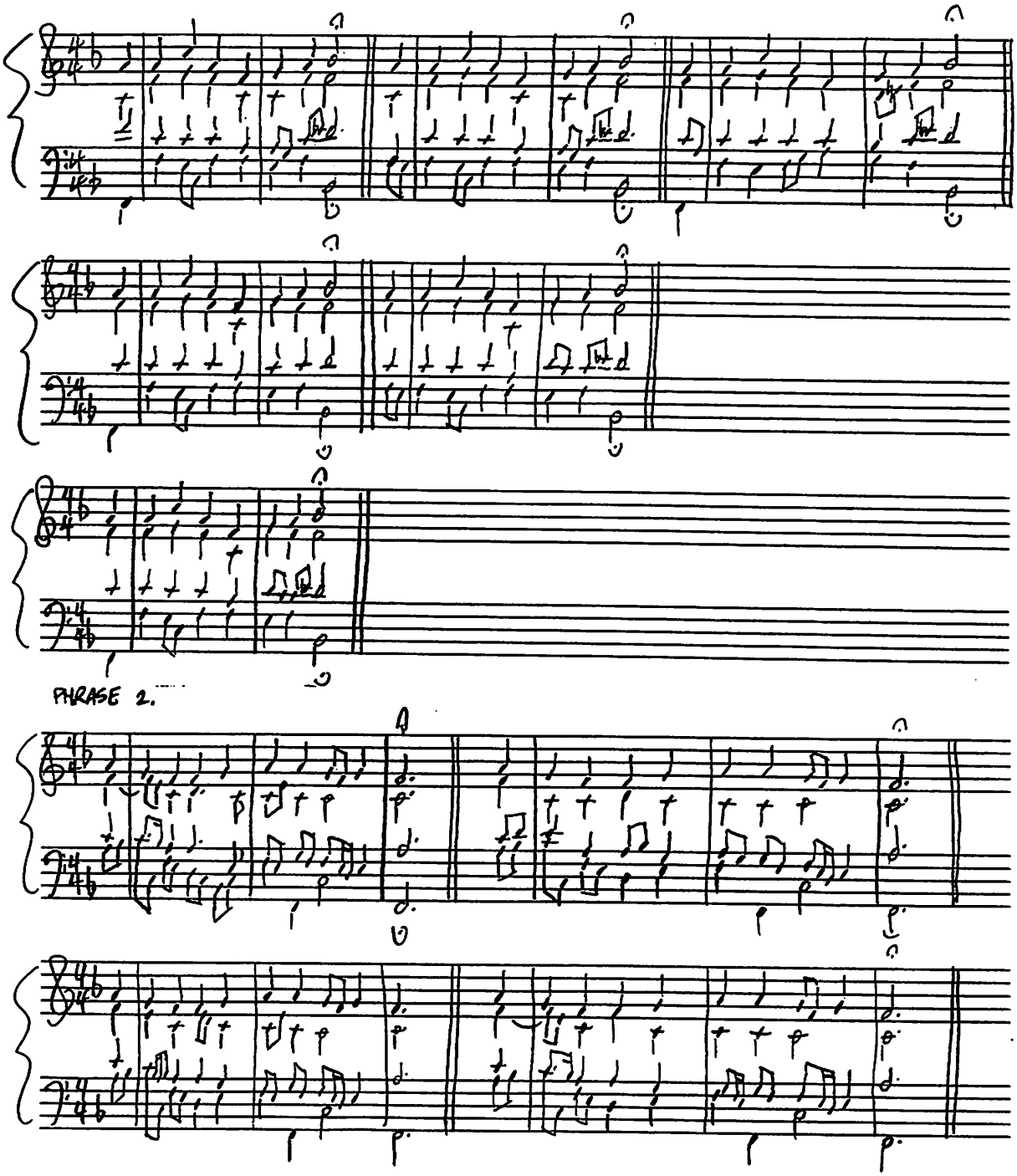
Phrase 3.



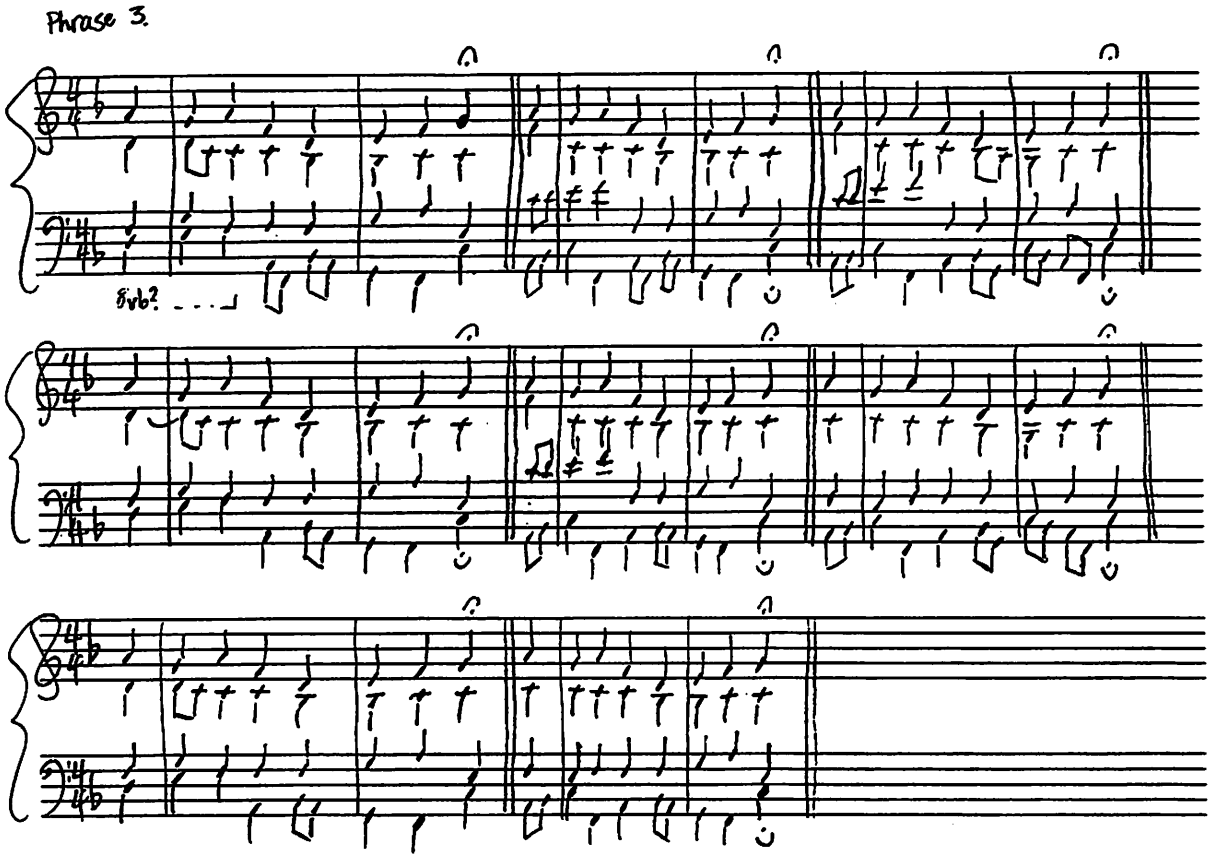
Phrase 4.



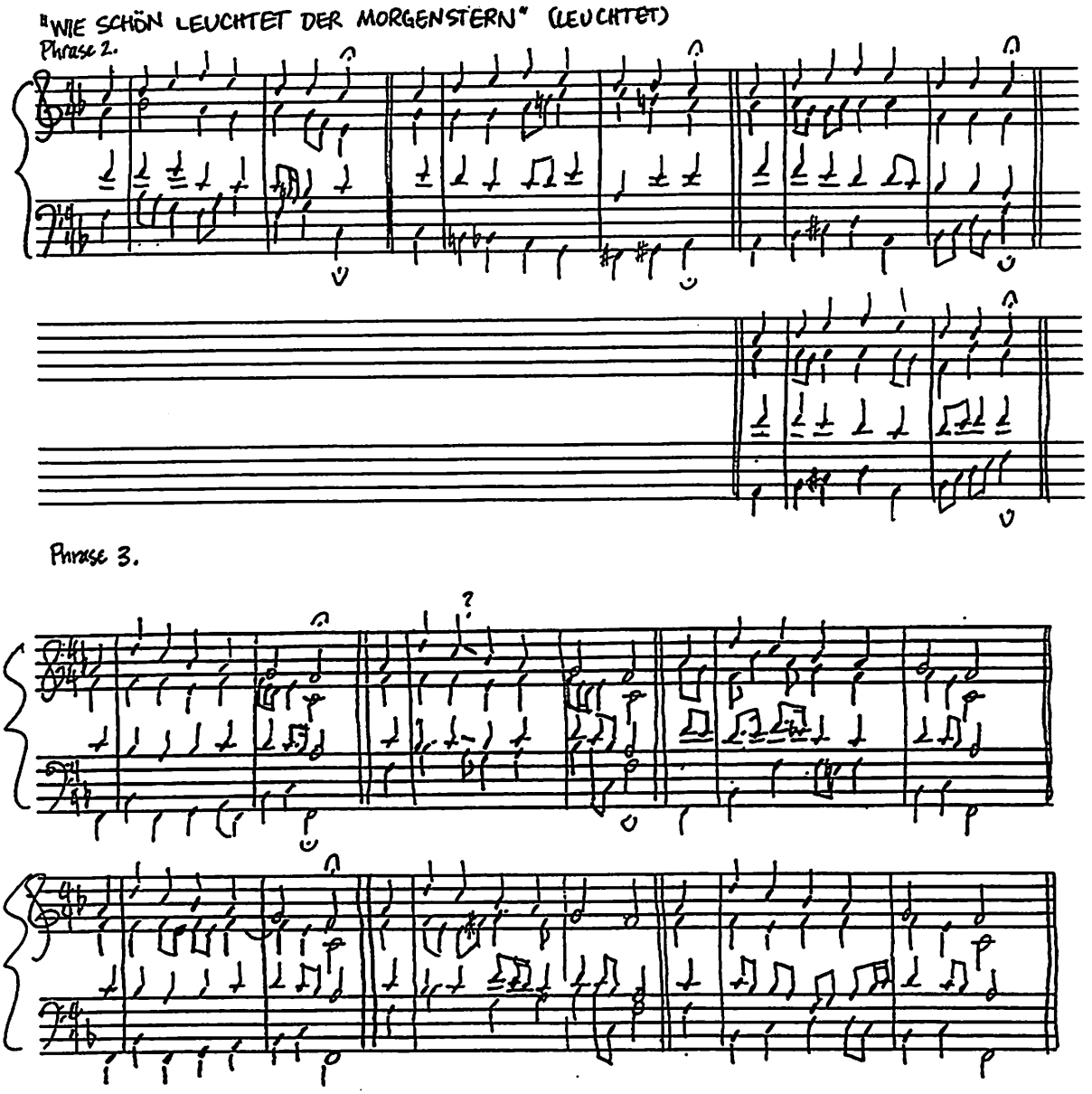
Melody: "Ach bleib bei uns, Herr Jesu Christ" (BLEIB)
Method C: The Top-Down Approach with metric information
Phrases 1 and 2
Examples BLEIB-C.1.a-f and BLEIB-C.2.a-d



Melody: "Ach bleib bei uns, Herr Jesu Christ" (BLEIB)
Method C: The Top-Down Approach with metric information
Phrase 3
Examples BLEIB-C.3.a-h

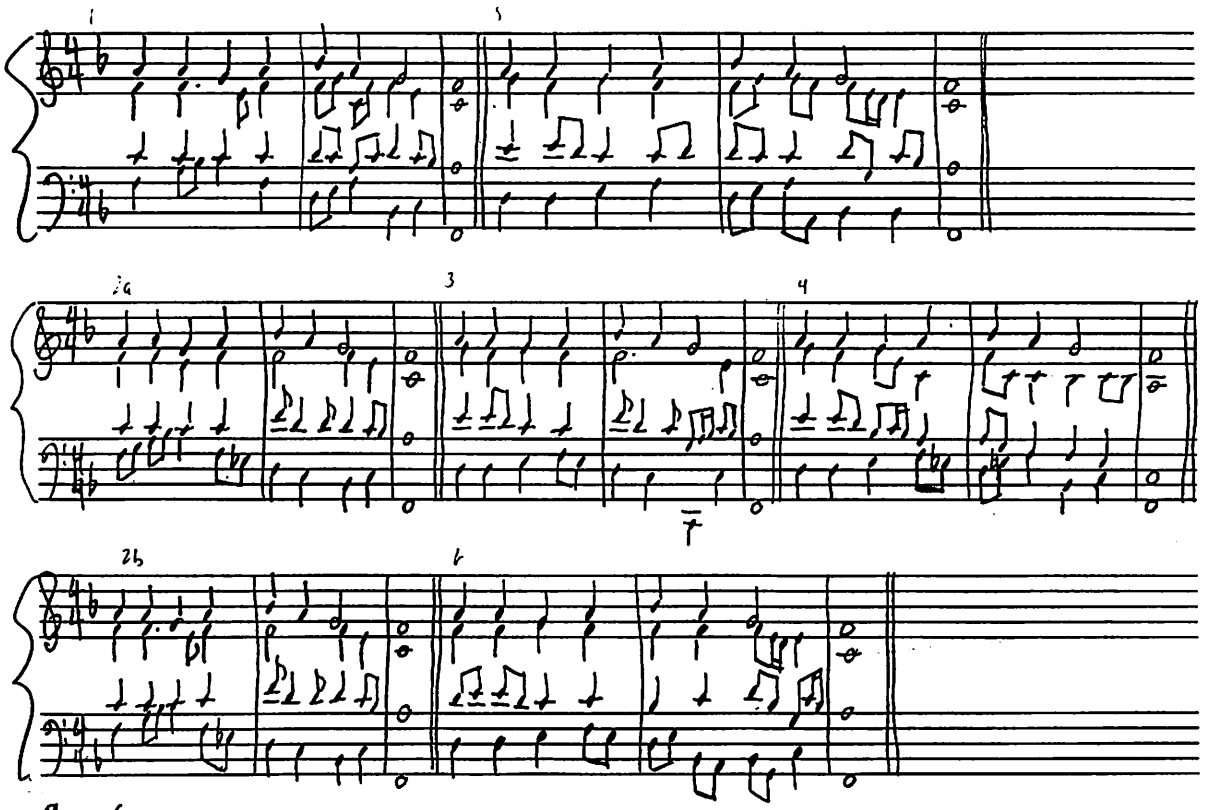


Melody: "Wie schön leuchtet der Morgenstern" (LEUCHTET)
Method C: The Top-Down Approach with metric information
Phrases 2 and 3
Examples LEUCHTET-C.2.a-d and LEUCHTET-C.3.a-f

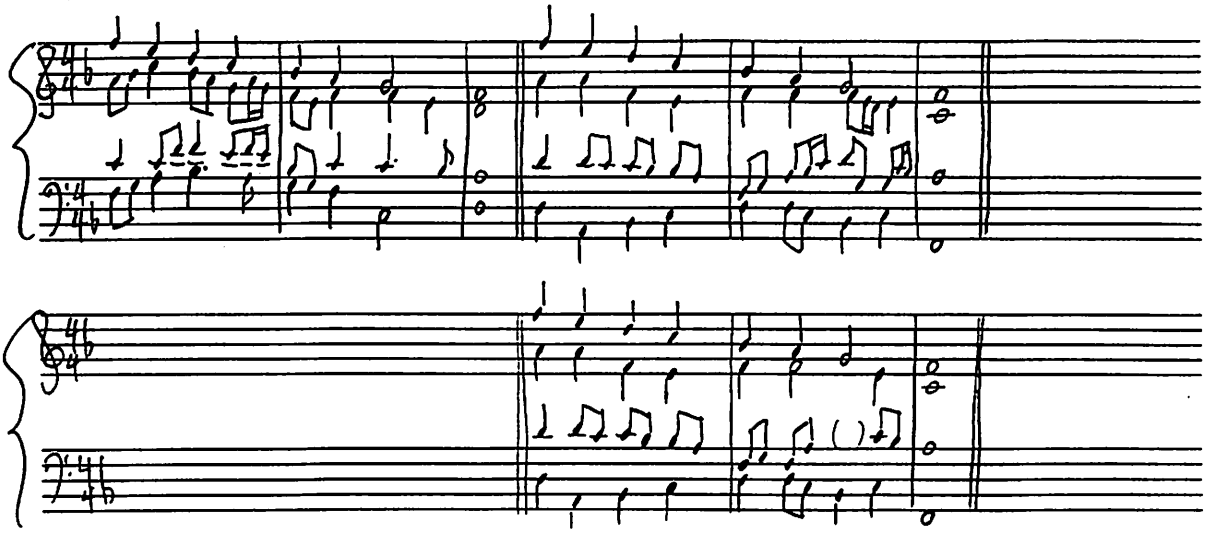


Melody: "Wie schön leuchtet der Morgenstern" (LEUCHTET)
Method C: The Top-Down Approach with metric information
Phrases 5 and 6
Examples LEUCHTET-C.5.a-g and LEUCHTET-C.6.a-c

Phrase 5.



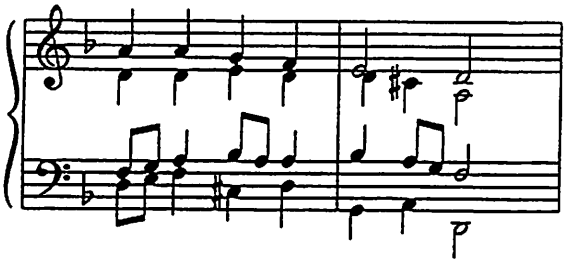
Phrase 6.



Appendix C

Data Format Examples

"Jesu, meine Freude"



The KERN data format

```
!!!COM: Bach, Johann Sebastian
!!!OTL: Jesu, meine Freude
**kern  **kern  **kern  **kern
*k[b-]  *k[b-]  *k[b-]  *k[b-]
*M4/4   *M4/4   *M4/4   *M4/4
=1-     =1-     =1-     =1-
8D\L    8F/L    4d/     4a/
8E\J    8G/J    .       .
4F\,    4A/,    4d/,    4a/,
4C#/    8B-/L   4e/     4g/
.       8A/J    .       .
4D\     4A/     4d/     4f/
=2      =2      =2      =2
4GG/    4B-\    4d/     2e;/
4AA/    8A/L    4c#/    .
.       8G/J    .       .
2DD;/,  2F;/,   2A;/,   2d;/,
```

The previous fragment in my LISP data format

```
(((|4| SO 0))  ((|4| DO 0))  ((|8| ME 0) (|8| FA 4))   ((|8| DO 0) (|8| RE 4)))
(((|4| SO 8))  ((|4| DO 8))  ((|4| SO 8))             ((|4| ME 8)))
(((|4| FA 16)) ((|4| RE 16)) ((|8| LE 16) (|8| SO 20)) ((|4| TI 16)))
(((|4| ME 24)) ((|4| DO 24)) ((|4| SO 24))            ((|4| DO 24)))
(((|2| RE 0))  ((|4| DO 0) (|4| TI 8)) ((|4| LE 0) (|8| SO 8) (|8| FA 12))
               ((|4| FA 0) (|4| SO 8)))
(((|2| DO 16)) ((|2| SO 16)) ((|2| ME 16))            ((|2| DO 16)))
((CADENCE))
(((|4| SO 0))  ((|4| DO 0))  ((|8| ME 0) (|8| RE 4))   ((|8| DO 0) (|8| RE 4)))
(((|4| LA 8))  ((|4| FA 8))  ((|4| DO 8))             ((|4| ME 8)))
(((|4| TE 16)) ((|8| FA 16) (|8| ME 20)) ((|4| TE 16)) ((|8| RE 16) (|8| ME 20)))
(((|4| SO 24)) ((|4| RE 24)) ((|4| TI 24))            ((|4| FA 24)))
(((|2| DO 0))  ((|4| SO 0) (|4| LA 8)) ((|8| DO 0) (|8| RE 4) (|4| ME 8))
               ((|8| MI 0) (|8| FI 4) (|4| FI 8)))
(((|2| TI 16)) ((|2| SO 16))  ((|2| RE 16))           ((|2| SO 16)))
((CADENCE))
```

Appendix D

Code Listing

chorale.h

This file contains data structures and headers for my C++ representation of a four-part chorale.

```
// chorale.h
//
// Christopher Thorpe
// 24 December 1997
//
// Classes and headers for chorale music data structures.
//

#ifndef _CHORALE_H_
#define _CHORALE_H_

#include "catlib.h"

const   int     QUANTUM  = 32;  // greatest subdivisions per whole note

const int SOL_FLAT    = 0;
const int SOL_NATURAL = 1;
const int SOL_SHARP   = 2;

class   pitch_t;

extern  const   pitch_t FIXED_DO;
extern  const   pitch_t EMPTY;
extern  const   pitch_t REST;

// Gets the pitch class of a note.
// c is middle C; cc one octave above; C one octave below and CC two below.
// octaves change at middle C, e.g. A B c d ... a b cc dd
class pitch_t {
  private:
```

```
    const int   MIDDLE_C = 51;

    char        letter;
    int         octave;
    int         sharp_or_flat;
    int         index;

    int         compute_index();  // helper in constructor

  public:
    pitch_t();
    pitch_t(char l, int o, int sf);

    int get_pc()    { return index % 12; } // returns pitch class (0 <= pc < 12)
    int get_index() { return index; }      // returns integer pitch index
    int get_octave() { return octave; }    // returns integer octave

    bool operator-(const pitch_t &p2);
    bool operator+(const pitch_t &p2);
    bool operator==(const pitch_t &p2);
    bool operator!=(const pitch_t &p2);

    // Allow a movable do
    const char *get_solfa(pitch_t use_do = FIXED_DO, bool chromatic = true);
};
pitch_t get_pitch(const char *s);

inline bool octave_equal(pitch_t p1, pitch_t p2)
{
    return p1.get_pc() == p2.get_pc();
}


typedef char duration_t;

class note_t {
  private:
    pitch_t     pitch;
    duration_t  duration;
    int         information;
    char        kern_description[32];
    char        kern_duration[8];

  public:
    note_t();
    note_t(pitch_t p, duration_t d, int i, const char *kd);

    pitch_t     get_pitch() { return pitch; }
    duration_t  get_duration() { return duration; }
    const char* get_kern_description() { return kern_description; }
    const char* get_kern_duration() { return kern_duration; }
    int         get_information() { return information; }

    const int   PAUSE      = 0x1;
```

```
    const int   BREATH     = 0x2;
    const int   TIE_BEGIN  = 0x4;
    const int   TIE_END    = 0x8;
    const int   TIE_WITHIN = 0x10;
};

note_t  get_note(const char *s);


// To represent Western 18th century key signatures, we use
// -1 for each flat, +1 for each sharp.  (Should range between [-7,+7])
// Examples: 0: C,a  +3: A/f#  -2: Bb/g
typedef int key_signature_t;

int compute_key_signature(const char *s);

// Enumerated type for the various modes.
// Since this is 17/18th century, not 9th, we use the
// modern theoretical modes (the white keys on the piano starting from C.)
typedef enum {Ionian=0, Dorian, Phrygian, Lydian, Mixolydian, Aeolian, Locrian}
        mode_t;
const   mode_t  Major = Ionian;
const   mode_t  Minor = Aeolian;
const   char *  print_mode(mode_t m);

// This way, by taking the negative or positive index of circle_of_*_keys
// you get the tonic note.   E.g. -3, 3 flats -> e- (E flat major)
// or 2, 2 sharps -> b (b minor.)
//
const pitch_t get_modal_tonic(key_signature_t key, mode_t mode);
const char * print_modal_tonic(key_signature_t key, mode_t mode);

inline const pitch_t get_major_tonic(key_signature_t key)
{ return get_modal_tonic(key, Major); }

inline const pitch_t get_minor_tonic(key_signature_t key)
{ return get_modal_tonic(key, Minor); }

int get_modal_pc(int degree, mode_t mode);

inline int get_major_pc(int degree) { return get_modal_pc(degree, Major); }
inline int get_minor_pc(int degree) { return get_modal_pc(degree, Minor); }

class time_signature_t {
  private:
    int         beat_duration;
    int         beats_per_measure;

    char        name_array[16];

  public:
    time_signature_t();
    time_signature_t(int b_d, int b_p_m);
    time_signature_t(time_signature_t &ts);
```

```
    const char *print();

    int         how_strong(int beat);

    int         get_beat_duration() { return beat_duration; }
    int         get_beats_per_measure() { return beats_per_measure; }
    int         get_measure_duration()
                        { return beat_duration * beats_per_measure; }
};
time_signature_t compute_time_signature(const char *s);

////////////////////
// Unique to chorales


const int MAX_BEATS = 6;

class chorale_t {
  public:
    enum        { BASS = 0, TENOR = 1, ALTO = 2, SOPRANO = 3 };
    const int   MAX_VOICES   = 4;
    const int   MAX_MEASURES = 48;  // maximum number of measures
    const int   MAX_LENGTH   = 6;   // maximum number of beats per measure
    const int   MAX_QUANTA   = (int) (QUANTUM * MAX_LENGTH/4.0 * MAX_MEASURES);

    chorale_t(time_signature_t &ts, int pickup, int measures,
                key_signature_t ks, mode_t m = Major, int voices = 4);
    chorale_t(const char *filename);

    note_t      get_event(int v, int p) { return the_music[v][p]; }

    key_signature_t     get_key_signature() { return key_signature; }
    time_signature_t    get_time_signature() { return time_signature; }
    mode_t              get_mode() { return mode; }
    pitch_t             get_tonic(){return get_modal_tonic(key_signature,mode);}

    const int   get_num_pickup_quanta() { return num_pickup_quanta; }
    const int   get_num_quanta() { return num_quanta; }
    const int   get_num_measures() { return num_measures; }
    const int   get_num_voices() { return num_voices; }
    const int   get_num_cadences() { return num_cadences; }

    const int   get_cadence(int i) { return cadences[i]; }

  private:
    time_signature_t time_signature;
    key_signature_t  key_signature;
    mode_t           mode;

    // Divide MAX_LENGTH by 4 to get
    note_t      the_music[MAX_VOICES][MAX_QUANTA];
    int         min_duration(int position); // the shortest event at position
```

```
    int         num_cadences;
    int         cadences[MAX_QUANTA/QUANTUM];

    int         num_pickup_quanta;
    int         num_quanta;
    int         num_measures;
    int         num_voices;
};

#endif // _CHORALE_H_
```

chorale.cc

This file contains function definitions for reading a file in the KERN format and creating the chorale data structure defined in chorale.h.

```
// chorale.cc
//
// Christopher Thorpe
// 26 December 1997
//
// Functions for chorale music data structures.
//
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <assert.h>

#include "chorale.h"

const   pitch_t FIXED_DO('c', 0, 0);
const   pitch_t EMPTY('x', 0, 0);
const   pitch_t REST('r', 0, 0);

const   char *  mode_name_data[7] = {"Ionian", "Dorian", "Phrygian", "Lydian",
                                     "Mixolydian", "Aeolian", "Locrian"};

const   char *
print_mode(mode_t m) {
    return mode_name_data[m];
}

const char* circle_data[21] = {
    "f-", "c-", "g-", "d-", "a-", "e-", "b-", "f",
    "c",
    "g", "d", "a", "e", "b", "f#", "c#", "g#", "d#", "a#", "e#", "b#" };

// The offsets are into the circle_data array if the key has no sharps or
// flats.  In any mode, adding a sharp or flat moves the same way across
// the circle of keys, so we keep track of the "starting point" for each
// mode (the note where the mode starts with no sharps or flats).
//
const int mode_identity_offsets[7] = { 8, 10, 12, 7, 9, 11, 13 };

const int  white_key_pcs[15] =
    { 0, 2, 4, 5, 7, 9, 11, 12, 14, 16, 17, 19, 21, 23, 24 };

int get_modal_pc(int degree, mode_t mode)
{
    return (white_key_pcs[mode + (degree % 7)]) - white_key_pcs[mode];
}

//////////////////////////////////////////
```

```
// This way, by taking the negative or positive index of circle_of_*_keys
// you get the tonic note.   E.g. -3, 3 flats -> e- (E flat major)
// or 2, 2 sharps -> b (b minor.)
//
const char *
print_modal_tonic(key_signature_t key, mode_t mode)
{
    int offset = mode_identity_offsets[mode] + key;
    assert (offset >= 0 && offset < 21);
    return (circle_data[offset]);
}

const pitch_t
get_modal_tonic(key_signature_t key, mode_t mode)
{
    int offset = mode_identity_offsets[mode] + key;
    assert (offset >= 0 && offset < 21);
    return get_pitch(circle_data[offset]);
}

pitch_t::pitch_t()
{
    letter = 'c';
    octave = 0;
    sharp_or_flat = 0;
    index = compute_index();
}

pitch_t::pitch_t(char l, int o, int sf)
{
    letter = l;
    octave = o;
    sharp_or_flat = sf;
    index = compute_index();
}

// Harsh equality.  For now, returns false for enharmonic equivalents.
bool pitch_t::operator== (const pitch_t &p2) {
    return (letter == p2.letter &&
            octave == p2.octave &&
            sharp_or_flat == p2.sharp_or_flat);
}

bool pitch_t::operator!= (const pitch_t &p2) {
    if (letter != p2.letter ||
            octave != p2.octave ||
            sharp_or_flat != p2.sharp_or_flat) {
                printf("Foo!");
    }
    return (letter != p2.letter ||
            octave != p2.octave ||
            sharp_or_flat != p2.sharp_or_flat);
}
```

```
const char *chr_solfa[7][3] = {
    { "de", "do", "di" },
    { "ra", "re", "ri" },
    { "me", "mi", "my" },
    { "fe", "fa", "fi" },
    { "se", "so", "si" },
    { "le", "la", "li" },
    { "te", "ti", "ty" }
};

const char *
pitch_t::get_solfa(pitch_t use_do = FIXED_DO, bool chromatic = true)
{
    int degree, accidental;

    if (letter == 'r') {
        return "r";
    }
    if (letter == 'x') {
        return "x";
    }
    if (index < 0) {
        return "x";
    }

    degree = positive_mod(letter - use_do.letter, 7);
    if (chromatic) {
        accidental = positive_mod(index-use_do.index, 12)-get_major_pc(degree);
    } else {
        accidental = 0;
    }
    return chr_solfa[degree][accidental + 1]; // +1 because 0 flat, 1 nat, etc.

}

pitch_t get_pitch(const char *s)
{
    int letter, octave, sharp_or_flat;

    letter = tolower(*s);
    octave = 0;
    sharp_or_flat = 0;

    // If it's a musical note, then compute the correct octave,
    // sharps or flats, etc.
    if (letter >= 'a' && letter <= 'g') {

        // If uppercase, go down octaves.
        while (isupper(*s)) {
            s++;
            octave--;
        }

        // Add an octave for each extra lowercase letter
```

```
        if (islower(*s)) {
            s++;
            while (islower(*s)) {
                s++;
                octave++;
            }
        }
        while (*s == '#') {
            s++;
            sharp_or_flat++;
        }
        while (*s == '-') {
            s++;
            sharp_or_flat--;
        }

        if (*s != '\0') {
            fprintf(stderr, "Garbage at end of pitch definition: %s\n", s);
        }

    }
    pitch_t result(letter, octave, sharp_or_flat);

    return result;
}

int pitch_t::compute_index()
{
    int result = MIDDLE_C;

    switch (letter) {
      case 'c':
        break;
      case 'd':
        result += 2;
        break;
      case 'e':
        result += 4;
        break;
      case 'f':
        result += 5;
        break;
      case 'g':
        result += 7;
        break;
      case 'a':
        result += 9;
        break;
      case 'b':
        result += 11;
        break;
      case 'r':
        result = -1;
        return result;  // don't do extra processing on this!
```

```
      default:
        result = -2;
        return result;  // don't do extra processing on this!
    }

    return (result + (12 * octave) + sharp_or_flat);
}

note_t::note_t(pitch_t p, duration_t d, int i, const char *kd)
{
    pitch = p;
    duration = d;
    information = i;
    strncpy(kern_description, kd, 31);
    if (isdigit(*kd)) {
        int i = 1;
        kern_duration[0] = *kd++;
        while ((isdigit(*kd) || (*kd == '.')) && i < 7) {
            kern_duration[i++] = *kd++;
        }
        kern_duration[i] = '\0';
    }
}

note_t::note_t()
{
    pitch = EMPTY;
    duration = 0;
    information = 0; // -1 is all bits set.  Not good. :]
    strcpy(kern_description, ".");
    strcpy(kern_duration, ".");
}

// -1 for each flat; +1 for each sharp.
int compute_key_signature(const char *s)
{
    int result;
    char buffer[64];
    char *t = buffer;

    s++;
    while (*s != ']') {
        *t++ = *s++;
    }
    *t = '\0';

    result = strlen(buffer)/2;
    if (result != 0 && buffer[1] == '-') {
        result = -result;
    }
    return result;
}

static const int beat_strengths[MAX_BEATS][MAX_BEATS] =
```

```
{
   { 1, 0, 0, 0, 0, 0 },
   { 1, 2, 0, 0, 0, 0 },
   { 1, 3, 2, 0, 0, 0 },
   { 1, 3, 2, 4, 0, 0 },
   { 1, 3, 2, 5, 4, 0 },
   { 1, 5, 3, 2, 6, 4 }
};

inline int time_signature_t::how_strong(int beat)
{
    return beat_strengths[beats_per_measure][beat];
}

chorale_t::chorale_t(time_signature_t &ts, int pickup, int measures,
                     key_signature_t ks = 0, mode_t m = Major, int voices=4)
{
    time_signature    = ts;
    num_pickup_quanta= pickup;
    num_measures      = measures;
    key_signature     = ks;
    mode              = m;
    num_voices        = voices;
    num_cadences      = 0;
}

// Figures out how many quanta in a note
static int bitreverse(int data, int quantum)
{
    int times = 0;
    while (quantum > 1) {
        quantum >>= 1;
        times++;
    }

    while (data > 1 && times-- > 0) {
        data >>= 1;
    }
    while (times-- > 0) {
        data <<= 1;
    }
    return data;
}

time_signature_t compute_time_signature(const char *s)
{
    int beats_per_measure, beat_duration;

    char buffer[16];
    char *t = buffer;

    while (isdigit(*s)) {
        *t++ = *s++;
    }
```

```
    *t = '\0';

    beats_per_measure = atoi(buffer);

    if (*s++ != '/') {
        fprintf(stderr, "Invalid string in time signature: %s\n", s);
        exit(1);
    }

    t = buffer;
    while (isdigit(*s)) {
        *t++ = *s++;
    }
    *t = '\0';

    beat_duration = zdiv(QUANTUM, atoi(buffer));
    time_signature_t result(beat_duration, beats_per_measure);
    return result;
}

const char * time_signature_t::print()
{
    return name_array;
}

time_signature_t::time_signature_t()
{
    beat_duration = -1;
    beats_per_measure = -1;
    sprintf(name_array, "[none]");
}

time_signature_t::time_signature_t(int b_d, int b_p_m)
{
    beat_duration = b_d;
    beats_per_measure = b_p_m;
    sprintf(name_array, "%d/%d", beats_per_measure,
                                 zdiv(QUANTUM, beat_duration));
}

time_signature_t::time_signature_t(time_signature_t &ts)
{
    beat_duration = ts.beat_duration;
    beats_per_measure = ts.beats_per_measure;
    sprintf(name_array, "%d/%d", beats_per_measure,
                                 zdiv(QUANTUM, beat_duration));
}

// Convert a string into a pitch/duration combination
note_t get_note(const char *s)
{
    char temp_buf[64];
    char *t = temp_buf;
    const char *kd= s;
```

```
int information = 0, dot_duration;
duration_t duration;

// Blank entry (single dot)
if (*s == '.' && !isgraph(*(s+1))) {
    note_t result(EMPTY, 0, 0, kd);
    return result;
}

if (*s == '[') {
    information |= note_t::TIE_BEGIN;
    s++;
}

// read number
while (isdigit(*s)) {
    *t++ = *s++;
}
*t = '\0';
duration = zdiv(QUANTUM, atoi(temp_buf));

// deal with the dots
dot_duration = duration/2;
while (*s == '.') {
    s++;
    duration += dot_duration;
    dot_duration /= 2;  // each extra dot is another half
}

// read note
t = temp_buf;
while (isalpha(*s) || *s == '#' || *s == '-') {
    *t++ = *s++;
}
*t = '\0';
pitch_t pitch = get_pitch(temp_buf);

// read any extra information (for now, just cadence points and tie ends)
while (*s != '\0' && *s != '\n') {
    switch (*s) {
      case '_':
        information |= note_t::TIE_WITHIN;
        break; // from switch
      case ']':
        information |= note_t::TIE_END;
        break; // from switch
      case ';':
        information |= note_t::PAUSE;
        break; // from switch
      case ',':
        information |= note_t::BREATH;
        break; // from switch
      default:
        break; // from switch
```

```
        }
        s++;
    }
    note_t result(pitch, duration, information, kd);
    return result;
}

static int
compute_num_voices(const char *s)
{
    int result = 0;
    while (*s != '\0') {
        if (isgraph(*s)) {
            result++;
            while (isgraph(*s)) {
                s++;
            }
        }
        while (isspace(*s)) {
            s++;
        }
    }
    return result;
}

static bool
kern_parse_line(const char *s, char notes[chorale_t::MAX_VOICES][16])
{
    int voice;
    char *tmp;

    // Advance past any initial space
    while (isspace(*s)) {
        s++;
    }
    if (*s == '=' || *s == '*' || *s == '!') {
        return false;
    }

    for (voice = 0; voice < chorale_t::MAX_VOICES; voice++) {
        tmp = notes[voice];

        // Advance past any space
        while (isspace(*s)) {
            s++;
        }

        // Copy in the good stuff
        while ((!isspace(*s)) && (s != '\0')) {
            *tmp++ = *s++;
        }
        *tmp = '\0'; // attach string terminator
    }
    return true;
```

```
}

int
chorale_t::min_duration(int position)
{
    int voice, current, result = -1;

    for (voice = 0; voice < num_voices; voice++) {
        current = the_music[voice][position].get_duration();
        if (current != 0 && (result == -1 || current < result)) {
            result = current;
        }
    }
    return result;
}

// This function assumes the kern format.
// It also assumes four spines in the order B T A S
// with key and time signatures appearing at the top of each spine.
//
chorale_t::chorale_t(const char *filename)
{
    char buffer[257];           // 256 should be long enough for any line

    FILE *fp;

    // For the notes for each voice
    char notes[MAX_VOICES][16];
    int  next_quantum[MAX_VOICES];

    int  q, lines = 0, measure_length = 0;
    int  current_measure = 0;

    num_cadences = 0;

    for (int i = 0; i < MAX_VOICES; i++) {
        next_quantum[i] = 0;
    }

    fp = fopen(filename, "r");
    if (fp == NULL) {
        fprintf(stderr, "Error opening file %s.  Exiting.\n", filename);
        exit(1);
    }

    // Read through headers
    // Set time and key signatures
    while ((++lines) && fgets(buffer, 256, fp) != NULL) {
        if (buffer[0] == '!') { // comment
            continue;
        } else if (buffer[0] == '*') {
            if (buffer[1] == 'M') { // time signature
                num_voices = compute_num_voices(buffer);
                time_signature = compute_time_signature(&buffer[2]);
```

```
        } else if (buffer[1] == 'k') { // key signature
            key_signature = compute_key_signature(&buffer[2]);
        }
        // Ignore other instructions
        continue;
    } else {
        break; // Through with headers, so break out of this loop!
    }
}

// Check for pickup measure and set value
if (buffer[0] != '=') { // pickup measure
    while(kern_parse_line(buffer, notes)) {
        for (int v = 0; v < num_voices; v++) {
            note_t n = get_note(notes[v]);
            the_music[v][next_quantum[v]] = n;
            next_quantum[v] += n.get_duration();
        }
        fgets(buffer, 256, fp); lines++;
    }
}
if (buffer[0] != '=') { // something's wrong here!
    fprintf(stderr, "Error reading line %d of file %s.\n"
                    "Expected: measure delimiter.\nGot: %s\n",
                    lines, filename, buffer);
    exit(1);
}
for (int v = 1; v < num_voices; v++) {
    if (next_quantum[v-1] != next_quantum[v]) {
        fprintf(stderr, "Error reading line %d of file %s.\n"
                        "Pickup measure is irregular.\nGot: %s\n",
                        lines, filename, buffer);
        exit(1);
    }
}
num_pickup_quanta = next_quantum[0];

// Read each line of music.  Check that measure indicators match.
// We are on the first measure indicator, so read the next line.
while ((++lines) && fgets(buffer, 256, fp) != NULL) {
    if (buffer[0] == '*') {
        if (buffer[1] == '-') {
            break; // from while loop
        }
        continue; // ignore kern directives.
    }

    measure_length = 0;
    current_measure++;

    while (kern_parse_line(buffer, notes)) {
        for (int v = 0; v < num_voices; v++) {
            note_t n = get_note(notes[v]);
```

```
            // Check for cadences in the soprano
            if (v == SOPRANO && (n.get_information() & note_t::PAUSE)) {
                cadences[num_cadences++] = next_quantum[v];
            }
            the_music[v][next_quantum[v]] = n;
            next_quantum[v] += n.get_duration();
        }

        fgets(buffer, 256, fp); lines++;

        // Skip any kern information or partial measure breaks.
        while (buffer[0] == '=' || buffer[0] == '*') {
            if (buffer[0] == '=') {
                char *more;
                (void) strtol(&buffer[1], &more, 10);

                if (! isalpha(*more)) {
                    break; // break from this loop: a new measure number.
                }
            }
            fgets(buffer, 256, fp); lines++;
        }

    }

    // Make sure we got what we expected: a measure break
    if (buffer[0] != '=') {
        fprintf(stderr, "Error reading line %d of file %s.\n"
                        "Expected: measure delimiter.\nGot: %s\n",
                        lines, filename, buffer);
        exit(1);
    }

    // Check that the music agrees we should have a new measure
    for (int v = 1; v < num_voices; v++) {
        if (next_quantum[v-1] != next_quantum[v]) {
            fprintf(stderr, "Error reading line %d of file %s.\n"
                            "Measure is irregular.\nGot: %s\n",
                            lines, filename, buffer);
            exit(1);
        }
    }
    // If the measure is too short or too long, but not the last one
    if (((next_quantum[0] - num_pickup_quanta) %
            time_signature.get_measure_duration() != 0)
        && buffer[1] != '=') {
        fprintf(stderr, "Error reading line %d of file %s.\n"
                        "Measure is too short or too long.\nGot: %s\n",
                        lines, filename, buffer);
        exit(1);
    }

    // Continue onward, reading the next line of music
}
```

```
    // Set the number of total measures and quanta
    num_measures = current_measure;
    num_quanta   = next_quantum[0];

    // Set the mode of the piece from the bass note of the last chord.
    for(q = next_quantum[0];
        the_music[BASS][q].get_pitch() == EMPTY && q >= 0;
        q--) {
                // do nothing
    }

    mode_t m;
    for (m = Ionian; m <= Locrian; m = mode_t(m + 1)) {
        if (octave_equal(get_modal_tonic(key_signature, m),
                         the_music[BASS][q].get_pitch())) {
            mode = m;
            break; // from for loop
        }
    }

    // This function returns nothing, as it is a constructor.

}

void debug_output()
{
    time_signature_t ts(3,4);
    ts = compute_time_signature("4/4");
    ts = compute_time_signature("3/4");
    ts = compute_time_signature("12/16");
    ts = compute_time_signature("12/8");
    chorale_t foo(ts, 3, 24, 2);
    printf("%d\n", foo.MAX_QUANTA);

    chorale_t foo2("sample.krn");
    pitch_t tonic = get_modal_tonic(foo2.get_key_signature(), foo2.get_mode());


    for (int i = 0; i < 256; i+=4) {
        printf("%3d: Pitch: %-2d (%-3s)  Duration: %-2d  Information: %x\n",
                i,
                foo2.get_event(3, i).get_pitch().get_index(),
                foo2.get_event(3, i).get_pitch().get_solfa(tonic),
                foo2.get_event(3, i).get_duration(),
                foo2.get_event(3, i).get_information());
    }
}
```

markov.cc

This file reads in one or more chorales in the KERN file format using the functions in `chorale.cc` and outputs them as a series of events (as defined in Chapter 4).

```
// markov.cc
// Christopher Thorpe

#include "chorale.h"
#include <stdlib.h>
#include <stdio.h>

void show_usage()
{
    fprintf(stderr, "Usage:\n"
                    "markov [files]\n");
}

bool print_voice(chorale_t chorale, int voice, int q, int dur)
{
    int i = q;
    int last = dur + q;
    note_t n;
    bool cadence = false;

    do {
        n = chorale.get_event(voice, i);
        printf(" (|%s| %s %d)",
            n.get_kern_duration(),
            n.get_pitch().get_solfa(chorale.get_tonic()),
            (i - chorale.get_num_pickup_quanta()) %
                    chorale.get_time_signature().get_measure_duration());
        if (n.get_duration() == 0) {
            // Search for the next non-empty event
            do {
                i++;
            } while (((n = chorale.get_event(voice, i)).get_duration() == 0) &&
                     (i < last));
        } else {
            i += n.get_duration();
        }
        cadence = (n.get_information() & note_t::PAUSE);
    } while (i < last);
    return cadence;
}

void print_grams(chorale_t chorale)
{
    int q = 0, gramq;
    bool cadence;

    while (q < chorale.get_num_quanta()) {
```

```
        gramq = q;
        note_t n2 = chorale.get_event(chorale_t::SOPRANO, q);
        q += n2.get_duration();
        printf("(");
        printf("((|%s| %s %d))",
            n2.get_kern_duration(),
            n2.get_pitch().get_solfa(chorale.get_tonic()),
            (gramq - chorale.get_num_pickup_quanta()) %
                 chorale.get_time_signature().get_measure_duration());

        printf(" (");
        cadence = print_voice(chorale,chorale_t::ALTO,gramq,n2.get_duration());
        printf(") (");
        cadence = print_voice(chorale,chorale_t::TENOR,gramq,n2.get_duration());
        printf(") (");
        cadence = print_voice(chorale,chorale_t::BASS,gramq,n2.get_duration());
        printf("))\n");

        if (cadence && (n2.get_information() & note_t::PAUSE) &&
            ((n2.get_duration() + q - chorale.get_num_pickup_quanta()) %
                chorale.get_time_signature().get_beat_duration() == 0)) {
            printf("((CADENCE))\n");
        }
        gramq += n2.get_duration();
    }
}


int main(int argc, char **argv)
{
    if (argc <= 1) {
        show_usage();
        exit(1);
    }
    for(int i = 1; i < argc; i++) {
        chorale_t the_chorale(argv[i]);
        printf("%s:\n", argv[i]);
        print_grams(the_chorale);
    }
    exit(0);
}
```

chorale.lsp

This file contains the LISP functions used to harmonize a melody, as described in the text of this thesis. These functions assume output derived from that of the program in `markov.cc`.

```
; chorale.lsp
; Christopher Thorpe
;

(defvar *3-sets*)
(defvar *3sets*)
(defvar *strong-3sets*)
(defvar *strongest-3sets*)
(defvar *raw-data*)
(defvar *raw-data-reversed*)
(defvar *do-quarter*)
(defvar *SECOND-ORDER* T)

(defvar *QUANTA* 32)  ;; quanta per measure

;; Format
;; ngram ::= ( gram^n )
;; gram  ::= ( event(S) . ( event(B)* ) )
;; event ::= ( duration solfeg position )

(defvar *STRONG-FREQ* 16)
(defvar *FOUR-FOUR* T)

(defun strongest-beat (event)
  (= (third (first (get-soprano-fun event))) 0))

(defun strong-beat (event)
  (= (mod (third (first (get-soprano-fun event))) *STRONG-FREQ*) 0))

(defun babble ()
  (do ((result (reverse (random-elt (massoc '((CADENCE)) *3sets*
                                            :TEST #'EQUAL)))
               (cons (third (random-elt (12massoc (second result)
                                                  (first  result) *3sets*
                                                  :TEST #'PAIR-EQ)))
                     result)))
      ((equal '((CADENCE)) (car result)) (cdr result))))

(defun last2 (lst)
  (do ((ans lst (cdr ans)))
      ((null (cddr ans)) ans)))

(defun first3 (lst)
  (list (first lst) (second lst) (third lst)))

(defun first2 (lst)
  (list (first lst) (second lst)))
```

```
(defun rest-at (n lst)
  (if (= n 0)
      lst
      (cdr (rest-at (1- n) lst))))

(defun random-elt (lst)
  (if (null lst)
       NIL
       (nth (random (length lst)) lst)))

;; pitches are the same; rhythms don't matter
(defun pitch-eq (e1 e2)
  (if (or (atom e1) (atom e2))
      (eq e1 e2)
      (eq (cadr e1) (cadr e2))))

;; pitches and rhythms are the same
(defun note-eq (e1 e2)
  (if (or (atom e1) (atom e2))
      (eq e1 e2)
      (and (eq (car e1)  (car e2))
           (eq (cadr e1) (cadr e2)))))

;; pitches functionally the same
(defun tone-eq (lst1 lst2)
  (if (or (atom lst1) (atom lst2))
      (eq lst1 lst2)
      (eval `(or ,@(mapcar #'(lambda (x) (if(member x lst2 :test #'pitch-eq) T))
                           lst1)))))

;; unused
(defun pair-eq (e1 e2)
  (if (or (atom e1) (atom e2))
      (eq e1 e2)
      (and (note-eq (car e1) (car e2))
           (note-eq (cadr e1) (cadr e2)))))

;; pitches and rhythms the same, as well as strong or weak
(defun beat-eq (e1 e2)
  (if (or (atom e1) (atom e2))
      (eq e1 e2)
      (and (eq (first e1)  (first e2))
           (eq (second e1) (second e2))
           (= (mod (third e1) *STRONG-FREQ*) (mod (third e2) *STRONG-FREQ*)))))

;; pitches, rhythms, and position in measure the same
(defun position-eq (e1 e2)
  (if (or (atom e1) (atom e2))
      (eq e1 e2)
      (and (eq (first e1)  (first e2))
           (eq (second e1) (second e2))
           (=  (third e1)  (third e2)))))
```

```
(defvar *NOTE-TEST*)
(setf *NOTE-TEST* #'note-eq)

(defun eq-sop (quad1 quad2)
       (equal (car quad1) (car quad2)))

(defun make-chord  (so al te ba)  (list so al te ba))
(defmacro get-soprano (macdata) '(car ,macdata))
(defmacro get-alto    (macdata) '(cadr ,macdata))
(defmacro get-tenor   (macdata) '(caddr ,macdata))
(defmacro get-bass    (macdata) '(cadddr ,macdata))

(defun get-soprano-fun (data) (car data))
(defun get-alto-fun    (data) (cadr data))
(defun get-tenor-fun   (data) (caddr data))
(defun get-bass-fun    (data) (cadddr data))

(defun multi-harm7 (S-line &optional (len 3) &key (max 20) (final? NIL))
  (do ((ph (multi-phrase S-line
             :local-context-tests
                '(strong-any-eq-SandB strong-set-eq-SandB
                  strong-first-eq-SandB strong-eq-SandB)
             :midlevel-context-tests
                '(any-eq-SandB set-eq-SandB first-eq-SandB eq-SandB)
             :toplevel-context-tests
                '(any-eq-SandB set-eq-SandB first-eq-SandB eq-SandB)
             :cadence-len len :final? final?)
           (multi-phrase S-line
             :local-context-tests
                '(strong-any-eq-SandB strong-set-eq-SandB
                  strong-first-eq-SandB strong-eq-SandB)
             :midlevel-context-tests
                '(any-eq-SandB set-eq-SandB first-eq-SandB eq-SandB)
             :toplevel-context-tests
                '(any-eq-SandB set-eq-SandB first-eq-SandB eq-SandB)
             :cadence-len len :final? final?))
       (counter 0 (1+ counter)))
      ((or (not (member NIL ph :key #'get-bass-fun)) (>= counter max))
        (print ph)
        (print 'x)
        (terpri)
        (smooth-bass-line ph len :context-test #'strong-eq-SandB))))

(defun multi-harm6 (S-line &optional (len 3) &key (max 20) (final? NIL))
  (do ((ph (multi-phrase S-line
             :local-context-tests    '(strong-any-eq-SandB)
             :midlevel-context-tests '(any-eq-SandB)
             :toplevel-context-tests '(any-eq-SandB)
             :cadence-len len :final? final?)
           (multi-phrase S-line
             :local-context-tests    '(strong-any-eq-SandB)
             :midlevel-context-tests '(any-eq-SandB)
             :toplevel-context-tests '(any-eq-SandB)
             :cadence-len len :final? final?))
```

```
      (counter 0 (1+ counter)))
     ((or (not (member NIL ph :key #'get-bass-fun)) (>= counter max))
       (print ph)
       (print 'x)
       (terpri)
       (smooth-bass-line ph len :context-test #'strong-first-eq-SandB))))

(defun multi-harm5 (S-line &optional (len 3) &key (max 20) (final? NIL))
  (do ((ph (multi-phrase S-line
             :local-context-tests    '(strong-set-eq-SandB)
             :midlevel-context-tests '(set-eq-SandB)
             :toplevel-context-tests '(set-eq-SandB)
             :cadence-len len :final? final?)
           (multi-phrase S-line
             :local-context-tests    '(strong-set-eq-SandB)
             :midlevel-context-tests '(set-eq-SandB)
             :toplevel-context-tests '(set-eq-SandB)
             :cadence-len len :final? final?))
       (counter 0 (1+ counter)))
     ((or (not (member NIL ph :key #'get-bass-fun)) (>= counter max))
       (print ph)
       (print 'x)
       (terpri)
       (smooth-bass-line ph len :context-test #'strong-set-eq-SandB))))

(defun multi-harm4 (S-line &optional (len 3) &key (max 20) (final? NIL))
  (do ((ph (multi-phrase S-line
             :local-context-tests    '(strong-first-eq-SandB)
             :midlevel-context-tests '(first-eq-SandB)
             :toplevel-context-tests '(first-eq-SandB)
             :cadence-len len :final? final?)
           (multi-phrase S-line
             :local-context-tests    '(strong-first-eq-SandB)
             :midlevel-context-tests '(first-eq-SandB)
             :toplevel-context-tests '(first-eq-SandB)
             :cadence-len len :final? final?))
       (counter 0 (1+ counter)))
     ((or (not (member NIL ph :key #'get-bass-fun)) (>= counter max))
       (print ph)
       (print 'x)
       (terpri)
       (smooth-bass-line ph len :context-test #'strong-first-eq-SandB))))

(defun multi-harm3 (S-line &optional (len 3) &key (max 20) (final? NIL))
  (do ((ph (smooth-bass-line (multi-phrase S-line
                                           :local-context-tests '(eq-SandB)
                                           :cadence-len len :final? final?)
                                           len)
           (smooth-bass-line (multi-phrase S-line
                                           :local-context-tests '(eq-SandB)
                                           :cadence-len len :final? final?)
                                           len))
       (counter 0 (1+ counter)))
     ((or (not (member NIL ph :key #'get-bass-fun)) (>= counter max)) ph)))
```

```
(defun multi-harm2 (S-line &optional (len 3) &key (max 20) (final? NIL))
  (do ((ph (smooth-bass-line (multi-phrase S-line
                                          :cadence-len len :final? final?)
                                          len)
           (smooth-bass-line (multi-phrase S-line
                                          :cadence-len len :final? final?)
                                          len))
       (counter 0 (1+ counter)))
      ((or (not (member NIL ph :key #'get-bass-fun)) (>= counter max)) ph)))

(defun multi-harm (S-line &optional (len 3) &key (max 20) (final? NIL))
  (do ((ph (add-inner-voices (smooth-bass-line (multi-phrase S-line
                                          :cadence-len len :final? final?)
                                          len))
           (add-inner-voices (smooth-bass-line (multi-phrase S-line
                                          :cadence-len len :final? final?)
                                          len)))
       (counter 0 (1+ counter)))
      ((or (not (member NIL ph)) (>= counter max)) ph)))

(defun harm3 (S-line &optional (len 3) &key (max 20) (final? NIL))
  (do ((ph (regular-phrase S-line :cadence-len len :final? final?
            :chord-test #'strong-eq-SandB)
           (regular-phrase S-line :cadence-len len :final? final?
            :chord-test #'strong-eq-SandB))
       (counter 0 (1+ counter)))
      ((or (not (member NIL ph)) (>= counter max)) ph)))

(defun harm2 (S-line &optional (len 3) &key (max 20) (final? NIL))
  (do ((ph (regular-phrase S-line :cadence-len len :final? final?)
           (regular-phrase S-line :cadence-len len :final? final?))
       (counter 0 (1+ counter)))
      ((or (not (member NIL ph)) (>= counter max)) ph)))

(defun harm (S-line &optional (len 3) &key (max 20) (final? NIL))
  (do ((ph (add-inner-voices (regular-phrase S-line
                                          :cadence-len len :final? final?))
           (add-inner-voices (regular-phrase S-line
                                          :cadence-len len :final? final?)))
       (counter 0 (1+ counter)))
      ((or (not (member NIL ph)) (>= counter max)) ph)))

(defun regular-phrase (S-line &key (final? NIL) (order 2) (cadence-len 3)
                           (cascade T)  (sets *3sets*) (chord-test #'eq-SandB))
  (do* ((output (gen-cadence (reverse S-line) :len cadence-len :final? final?)
                (let ((new-chord (gen-chord (car S-curr) (first2 output)
                                            :order order :sets sets
                                            :chord-test chord-test)))
                     (if (or new-chord (not cascade))
                         (cons new-chord output)
                         (cons (gen-chord (car S-curr) (first2 output)
                                          :order (1- order) :sets sets
                                          :chord-test chord-test) output))))
```

```
        (S-curr (rest-at cadence-len (reverse S-line))
                (cdr S-curr)))
      ((null S-curr) output)))


(defun add-inner-voices (phrase &key (order 2) (cascade T) (sets *3sets*))
  (do* ((reversed-phrase (reverse phrase))
        (ph (if (= order 2)
                (cddr reversed-phrase)
                (cdr reversed-phrase))
            (cdr ph))
        (answer (if (= order 2)
                    (list (second reversed-phrase) (first reversed-phrase))
                    (list (first reversed-phrase)))))

       ((null ph) answer)

       (let ((new-chord (fill-chord (first2 answer) (car ph)
                                    :order order :sets sets)))
            (setq answer
                  (if (or new-chord (not cascade))
                      (cons new-chord answer)
                      (cons (fill-chord (first2 answer) (car ph)
                                        :order 1 :sets sets) answer))))))

(defun fill-chord (last next &key (order 2) (sets *3sets*))
  (do* ((ps (if (= order 2)
                (12massoc (second last) (first last) sets :test #'eq-harmony)
                (massoc (first last) sets :test #'eq-harmony))
            (cdr ps))

       (new-chord (if (= order 2) (third (first ps)) (second (first ps)))
                  (if (= order 2) (third (first ps)) (second (first ps))))

       (answer NIL))
      ((null ps) (if (null answer) NIL
                     (let ((a (random-elt answer)))
                      (setf (get-soprano a) (get-soprano next))
                      (setf (get-bass a) (get-bass next))
                      a)))
      (if (funcall #'eq-SandB new-chord next)
          (setq answer (cons new-chord answer)))))


(defun gen-cadence (S-line &key (len 3) (final? NIL) (test *NOTE-TEST*))
  (let ((result (random-elt
                 (gen-cadence-guts S-line :len len :final? final? :test test))))
       (nreverse (mapcar #'rplaca (nreverse result) (mapcar #'list s-line)))))

(defun gen-cadence-guts (S-line &key (len 3) (final? NIL) (test #'note-eq))
;  {Randomly choose from raw-data a (three?)-note-long sequence with pitch
;   classes in the soprano line matching those in S-line}
;
```

```
  (do ((found-list NIL)
       (curr-data (cdr *RAW-DATA-REVERSED*)
                  (cdr (member '((CADENCE)) curr-data :test #'equal))))
      ((null curr-data) found-list)
      (setq found-list
        (if (eval
            `(and ,(funcall #'pitch-eq (first S-line)  ; pitch-eq because the
                                                       ; the last line is held
                           (first (get-soprano (first curr-data))))
                  ,(or (not final?) ; if we're at a final cadence, bass is DO
                       (pitch-eq (first (get-bass (first curr-data)))
                                 *DO-QUARTER*))
                  ,@(mapcar #'(lambda (x y) (funcall test x
                                                     (first (get-soprano y))))
                           (subseq S-line 1 len)
                           (subseq curr-data 1 len))))
          ; reversed here because the data are reversed
          (cons (reverse (subseq curr-data 0 len)) found-list) found-list))))

;; gen-chord
  ;; It may become necessary to add a check here and choose different values
  ;; based on the length of the phrase.  It seems a more straightforward
  ;; implementation to take care of the beginning of the phrase in this
  ;; function than in the regular-phrase function, though the merit of the style
  ;; of this heirarchical decomposition is dubious.
  ;;
  ;; If the order argument is 1, this forces first-order probabilities;
  ;; and if 2 forces second-order.

(defun gen-chord (curr-sop last &key (test *NOTE-TEST*)
                                     (order 2)
                                     (chord-test #'eq-SandB)
                                     (sets *3sets*))
;  {Look at the last two (or appropriate order) entries contained in last2.
;   Find an instance of these two chords in progression, followed by the
;   soprano's next note (car S-curr) in any of the raw-data.  If it cannot
;   find an appropriate harmonization, use the first-order function.  Returns
;   the chord as a quadruple of triples.}

    (do* ((ps (if (= order 2)
                  (12massoc (second last) (first last) sets :test chord-test)
                  (massoc (first last) sets :test chord-test))
              (cdr ps))

          (new-chord (if (= order 2) (third (first ps)) (second (first ps)))
                     (if (= order 2) (third (first ps)) (second (first ps))))

          (answer NIL))
         ((null ps) (random-elt answer))
         (if (funcall test (first (get-soprano new-chord)) curr-sop)
             (setq answer
                   (cons (make-chord (get-soprano new-chord) NIL NIL
                                     (get-bass new-chord))
                         answer)))))
```

```
;; Code to harmonize with new multi-level Markov method

;; Since multi-harmonization is destructive, create new
;; list structure for the result, including the cadence.
(defun create-structure (S-line &key (len 3) (final? NIL) (test *NOTE-TEST*))
  (do ((result (gen-cadence (reverse s-line) :len len :final? final? :test test)
               (cons (list (list (car curr-sop)) NIL NIL NIL) result))
       (curr-sop (rest-at len (reverse s-line)) (cdr curr-sop)))
      ((null curr-sop) result))) ;(nreverse result))))

(defun smooth-bass-line (structure cadence-len &key (order 2)
                         (sets *3sets*)
                         (equiv-test #'any-eq-SandB)
                         (context-test #'strong-eq-SandB))
  (do ((length (length structure))
       (pos cadence-len (1+ pos))
       (result (nreverse (copy-tree structure))))
      ((= length pos) (nreverse result))
      (setf (get-bass (nth pos result))
            (cond
             ; before-and-after context
;            ((get-bass (third (random-elt
;              (remove-if-not
;               #'(lambda (x)
;                         (funcall equiv-test (second x)
;                                              (nth pos result)))
;               (13massoc (nth (- pos 1) result) (nth (+ pos 1) result)
;                                                 sets :test context-test))))))
             ((get-bass (third (random-elt
               (remove-if-not
                #'(lambda (x)
                          (funcall equiv-test (third x)
                                               (nth pos result)))
                (12massoc (nth (- pos 2) result)
                          (nth (- pos 1) result)
                          sets :test context-test))))))
             ((get-bass (second (random-elt
               (remove-if-not
                #'(lambda (x)
                          (funcall equiv-test (second x)
                                               (nth pos result)))
                (massoc (nth (- pos 1) result)
                        sets :test context-test))))))
             ((get-bass (nth pos result))) ; last resort -- don't change it
            ))))



(defun multi-phrase (S-line &key
         (final? NIL) (order 2) (cadence-len 3)
         (cascade T)
```

```
        (strongest-sets *strongest-3sets*)
        (strong-sets *strong-3sets*)
        (complete-sets *3sets*)
        (toplevel-context-tests '(eq-SandB))
        (midlevel-context-tests '(eq-SandB))
        (local-context-tests    '(strong-eq-SandB)))
  (let* ((structure (create-structure S-line :len cadence-len :final? final?))
        (toplevel  (remove-if-not #'strongest-beat structure))
        (midlevel  (remove-if-not #'strong-beat structure))
        (last-strongest (first (last toplevel)))
        (last-strong    (first (last midlevel))))

       ; Make sure the last bass event in the toplevel has harmony
       (setf (get-bass last-strongest)
             (or (get-bass last-strongest)
                 (get-bass
                  (random-elt
                   (remove-if-not
                    #'(lambda (x) (note-eq(first(get-soprano x))
                                          (first(get-soprano last-strongest))))
                    (mapcar #'second
                            (massoc last-strong strong-sets
                                    :test (car midlevel-context-tests))))))
                 (get-bass
                  (random-elt
                   (remove-if-not
                    #'(lambda (x) (note-eq(first(get-soprano x))
                                          (first(get-soprano last-strongest))))
                    (mapcar #'second
                            (massoc last-strong strong-sets
                                    :test (or
                                           (cadr midlevel-context-tests)
                                           #'eq-SandB))))))))

       (fill-toplevel toplevel
                      strongest-sets :context-tests toplevel-context-tests
                      :order order :cascade cascade)
       (fill-out midlevel strong-sets :context-tests midlevel-context-tests)
       (fill-out structure complete-sets :context-tests local-context-tests)
       structure))

(defun fill-toplevel
      (structure sets &key (order 2) (cascade t) (context-tests '(eq-SandB)))
  (nreverse structure)

  ;; Get the first order elements of structure
  (if (and (= order 2)
           (member nil (subseq (mapcar #'get-bass-fun structure) 0 2)))
      (setf (get-bass (second structure))
            (copy-tree
             ; Try at least two of our context tests
             (or (get-bass (gen-chord (first (get-soprano (second structure)))
                                      (list (first structure))
                                      :chord-test (car context-tests)
```

```
                                :order 1 :sets sets :test #'pitch-eq))
                (get-bass (gen-chord (first (get-soprano (second structure)))
                                (list (first structure))
                                :chord-test (or (car context-tests)
                                                #'eq-SandB)
                                :order 1 :sets sets :test #'pitch-eq)))))

  (do* ((length (length structure))
        (pos order (1+ pos)))
       ((= length pos) (nreverse structure))
       (setf (get-bass (nth pos structure))
             (copy-tree
              ; Try our various methods for getting the bass
              (do* ((ord order (1- ord))
                    (result '(NIL)))
                   ((or (zerop order) (and (/= ord order) (not cascade))
                        (not (member NIL result :key #'get-bass-fun)))
                    result)
                   (do* ((test-fn context-tests (cdr test-fn)))
                        ((or (null test-fn)
                             (not (member NIL result :key #'get-bass-fun)))
                         result)
                        (setq result
                         (gen-chord
                              (first (get-soprano (nth pos structure)))
                              (reverse (subseq structure 0 pos))
                              :chord-test (car test-fn)
                              :order ord :sets sets :test #'pitch-eq))))))))

;; Assumes structure's NIL elements are immediately preceded
;; and followed by non-NIL elements.
;; Attempts to assign the NIL elements based on this context.
;;
(defun fill-out (structure sets &key (context-tests '(eq-SandB)))
  (nreverse structure)

  (do* ((pos (position NIL structure :test #'eq-bass)
             (position NIL structure :test #'eq-bass)))
       ((or (null pos)
            (< pos 1))
        (nreverse structure))
       (let ((new-chords
              (context-chord (first (get-soprano (nth pos structure)))
                             (if (> pos 1) (nth (- pos 2) structure))
                             (nth (- pos 1) structure)
                             (nth (+ pos 1) structure)
                             (nth (+ pos 2) structure)
                             sets :tests context-tests)))
             (if (null new-chords) (return-from fill-out NIL))
             (setf (get-bass (nth pos structure))
                   (copy-tree (get-bass (random-elt new-chords)))))))

;; context should begin 2 elements before desired pitch
;;
```

```
;; Recall sets are reversed!
(defun context-chord (soprano 2before 1before 1after 2after sets
                      &key (tests '(eq-SandB)))
  (let ((result))
       (do* ((test-fn tests (cdr test-fn)))
            ((or (null test-fn) result))
            (setf result
                  (remove-if-not
                   #'(lambda (x) (note-eq (first (get-soprano x)) soprano))
                   (mapcar #'second
                           (13massoc 1before 1after sets :test(car test-fn))))))
       (if result (return-from context-chord result))
       (do* ((test-fn tests (cdr test-fn)))
            ((or (null test-fn) result))
            (setf result
             (intersection
              (remove-if-not
               #'(lambda (x) (note-eq (first (get-soprano x)) soprano))
               (mapcar #'second (massoc 1before sets :test (car test-fn))))
              (remove-if-not
               #'(lambda (x) (note-eq (first (get-soprano x)) soprano))
               (mapcar #'first (2massoc 1after sets :test (car test-fn))))
              :TEST (car test-fn))))

       ;; If we've found a result or we can't ignore later context, return
       (if (or result 1after)
           (return-from context-chord result))

       (do* ((test-fn tests (cdr test-fn)))
            ((or (null test-fn) result))
            (setf result
                  (remove-if-not
                   #'(lambda (x) (note-eq (first (get-soprano x)) soprano))
                   (mapcar #'third
                           (12massoc 2before 1before sets
                                     :test (car test-fn))))))

       (if result (return-from context-chord result))

       (do* ((test-fn tests (cdr test-fn)))
            ((or (null test-fn) result))
            (setf result
                  (remove-if-not
                   #'(lambda (x) (note-eq (first (get-soprano x)) soprano))
                   (mapcar #'second
                           (massoc 1before sets :test (car test-fn))))))
       result))

(defun caddadr (x) (caddr (cadr x)))
(defun caddaddr (x) (caddr (caddr x)))
(defun caaddar (x) (caar (cddar x)))
(defun caadddr (x) (caar (cdddr x)))
(defun caddadddr (x) (caddr (cadddr x)))
```

```
(defun eq-bass (chord1 chord2)
  (equal (get-bass chord1)
         (get-bass chord2)))

(defun strongest-eq-SandB (chord1 chord2)
  (eq-SandB chord1 chord2 :soprano-test #'position-eq))

(defun strong-eq-SandB (chord1 chord2)
  (eq-SandB chord1 chord2 :soprano-test #'beat-eq))

(defun strong-any-eq-SandB (chord1 chord2)
  (any-eq-SandB chord1 chord2 :soprano-test #'beat-eq))

(defun strong-set-eq-SandB (chord1 chord2)
  (set-eq-SandB chord1 chord2 :soprano-test #'beat-eq))

(defun strong-first-eq-SandB (chord1 chord2)
  (first-eq-SandB chord1 chord2 :soprano-test #'beat-eq))

(defun eq-SandB (chord1 chord2 &key (bass-test #'pitch-eq)
                                    (soprano-test #'note-eq))
  (if (or (atom chord1) (atom chord2))
      (eq chord1 chord2)
      ; note-eq for soprano, pitch-eq for bass
      (eval `(and ,(funcall soprano-test (first (get-soprano chord1))
                                         (first (get-soprano chord2)))
                  ,@(mapcar bass-test (get-bass chord1)
                                      (get-bass chord2))))))

(defun any-eq-SandB (chord1 chord2 &key (bass-test #'pitch-eq)
                                    (soprano-test #'beat-eq))
  (if (or (atom chord1) (atom chord2))
      (eq chord1 chord2)
      ; note-eq for soprano, pitch-eq for bass
      (eval `(and ,(funcall soprano-test (first (get-soprano chord1))
                                         (first (get-soprano chord2)))
                  (or
                  ,@(mapcar #'(lambda (x)
                                (if (member x (get-bass chord1) :test bass-test)
                                    T NIL))
                            (get-bass chord2))
                  ,@(mapcar #'(lambda (x)
                                (if (member x (get-bass chord2) :test bass-test)
                                    T NIL))
                            (get-bass chord1)))))))

(defun set-eq-SandB (chord1 chord2 &key (bass-test #'pitch-eq)
                                    (soprano-test #'beat-eq))
  (if (or (atom chord1) (atom chord2))
      (eq chord1 chord2)
      (let ((shorter (if (< (length chord1) (length chord2)) chord1 chord2))
            (longer  (if (< (length chord1) (length chord2)) chord2 chord1)))
      ; note-eq for soprano, pitch-eq for bass
      (eval `(and ,(funcall soprano-test (first (get-soprano shorter))
```

```
                                        (first (get-soprano longer)))
                  ,@(mapcar #'(lambda (x)
                                (if (member x (get-bass longer) :test bass-test)
                                    T NIL))
                            (get-bass shorter)))))))

(defun first-eq-SandB (chord1 chord2 &key (bass-test #'pitch-eq)
                                          (soprano-test #'beat-eq))
  (if (or (atom chord1) (atom chord2))
      (eq chord1 chord2)
      ; note-eq for soprano, pitch-eq for bass
      (eval `(and ,(funcall soprano-test (first (get-soprano chord1))
                                         (first (get-soprano chord2)))
                  ,(funcall bass-test (first (get-bass chord1))
                                      (first (get-bass chord2)))))))

(defun eq-harmony (chord1 chord2)
  (if (or (atom chord1) (atom chord2))
      (eq chord1 chord2)
      ; note-eq means they have to have the same rhythms.
      ; tone-eq means they have to be functionally equal,
      ;         but may have non-chord-tone differences.
      (eval `(and ,@(mapcar #'note-eq (get-soprano chord1)
                            (get-soprano chord2))
                  ,(tone-eq (get-alto chord1)
                            (get-alto chord2))
                  ,(tone-eq (get-tenor chord1)
                            (get-tenor chord2))
                  ,(tone-eq (get-bass chord1)
                            (get-bass chord2))))))

(defun new3sets (lst)
  (do ((l lst (cdr l))
       (answer nil (cons (list (first l) (second l) (third l)) answer)))
      ((null (cddr l)) answer)))

(defun reversed3sets (lst)
  (do ((l lst (cdr l))
       (answer nil (cons (list (third l) (second l) (first l)) answer)))
      ((null (cddr l)) answer)))

(defun 3-sets (lst)
  (do ((sets (3-sets-guts lst) (cdr sets))
       (answer NIL))
      ((null sets) answer)
      (setq answer
            (if (or (member 'cadence (car sets))
                    (member 'final-cadence (car sets)))
                answer
                (cons (car sets) answer)))))

(defun 3-sets-guts (lst)
  (if (< (length lst) 3) NIL
      (cons (list (car lst) (cadr lst) (caddr lst)) (3-sets (cdr lst)))))
```

```
(defun 2massoc (elem lst &key (test #'eq))
  (do ((l lst (cdr l))
       (answer nil
               (if (funcall test elem (cadar l))
                   (cons (car l) answer)
                   answer)))
      ((null l) answer)))

(defun massoc (elem lst &key (test #'eq))
  (do ((l lst (cdr l))
       (answer nil
               (if (funcall test elem (caar l))
                   (cons (car l) answer)
                   answer)))
      ((null l) answer)))

;  (cond ((null lst) NIL)
;        ((apply test `(,elem ,(caar lst)))
;         (cons (car lst) (massoc elem (cdr lst) :test test)))
;        (T
;         (massoc elem (cdr lst) :test test))))

(defun 12massoc (elem1 elem2 lst &key (test #'eq))
  (do ((l lst (cdr l))
       (answer nil
               (if (and (funcall test elem1 (caar l))
                        (funcall test elem2 (cadar l)))
                   (cons (car l) answer)
                   answer)))
      ((null l) answer)))
;
;  (cond ((atom lst) NIL)
;        ((atom (car lst)) NIL)
;        ((atom elem1) NIL)
;        ((atom elem2) NIL)
;        ((and (funcall test elem1 (caar lst))
;              (funcall test elem2 (cadar lst)))
;         (cons (car lst) (12massoc elem1 elem2 (cdr lst) :test test)))
;        (T
;         (12massoc elem1 elem2 (cdr lst) :test test))))

; (defun 13massoc (elem1 elem3 lst &key (test #'eq) (key #'car)
;  (cond ((null lst) NIL)
;        ((and (apply test `(,elem1 ,(caar lst)))
;              (apply test `(,elem3 ,(caddar lst))))
;         (cons (car lst) (13massoc elem1 elem3 (cdr lst) :test test)))
;        (T
;         (13massoc elem1 elem3 (cdr lst) :test test))))

(defun 13massoc (elem1 elem3 lst &key (test #'eq))
  (do ((l lst (cdr l))
       (answer nil
               (if (and (funcall test elem1 (caar l))
```

```
                    (funcall test elem3 (caddar l)))
                (cons (car l) answer)
                answer)))
      ((null l) answer)))

(defun 123massoc (elem1 elem2 elem3 lst &key (test #'eq))
  (do ((l lst (cdr l))
       (answer nil
               (if (and (funcall test elem1 (first  (car l)))
                        (funcall test elem2 (second (car l)))
                        (funcall test elem3 (third  (car l))))
                   (cons (car l) answer)
                   answer)))
      ((null l) answer)))
```

About the Recorded Examples

The 2-part examples on the tape are recorded in the order they appear in the text of the thesis. Not all examples in the text are recorded; these are provided as an aid for those reading the thesis. Each phrase is recorded with a pause before the beginning of the next phrase.

Following the 2-part examples I have recorded two full four-part chorales as harmonized by the method described in Appendix A.

List of Recorded Examples

Two-Part

1. LEUCHTET-I.1.a
2. LEUCHTET-I.1.b
3. LEUCHTET-I.3.b
4. LEUCHTET-I.3.c
5. BLEIB-I.4.g
6. BLEIB-I.3.a
7. BLEIB-I.3.b
8. LEUCHTET-I.2.c
9. BLEIB-I.1.g
10. BLEIB-I.4.f
11. BLEIB-I.4.d
12. BLEIB-II.1.d
13. BLEIB-II.1.e
14. BLEIB-II.1.f
15. LEUCHTET-III.3.c
16. LEUCHTET-III.3.e
17. LEUCHTET-IV.1.e
18. LEUCHTET-V.1.e

Four-Part

19. BLEIB-C.1.a
20. BLEIB-C.2.a
21. BLEIB-C.3.a
22. BLEIB-A.4.a
23. LEUCHTET-C.2.b
24. LEUCHTET-C.3.e
25. LEUCHTET-C.5.g
26. LEUCHTET-C.6.b